Introduction to Programming Using Java

Version 7.0, August 2014

(Version 7.0.2, with just a few corrections, December 2016 )

David J. Eck

Hobart and William Smith Colleges

This is a PDF version of a free, on-line book that is available

at http://math.hws.edu/javanotes/. The web site includes

source code for all example programs, answers to quizzes,

and discussions and solutions for exercises.

ii

c 1996–2016, David J. Eck

David J. Eck (eck@hws.edu)

Department of Mathematics and Computer Science

Hobart and William Smith Colleges

Geneva, NY 14456

This book can be distributed in unmodified form for non-commercial purposes. Modified versions can be made and distributed for non-commercial purposes provided they are distributed under the same license as the original. More specifically: This work is licensed under the Creative Commons Attribution NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/. Other uses require permission from the author.

The web site for this book is: http://math.hws.edu/javanotes

Contents

Preface xiii

1 The Mental Landscape 1 1.1 Machine Language . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1 1.2 Asynchronous Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3 1.3 The Java Virtual Machine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6 1.4 Building Blocks of Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8 1.5 Object-oriented Programming . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10 1.6 The Modern User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13 1.7 The Internet and Beyond . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15 Quiz on Chapter 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

2 Names and Things 19 2.1 The Basic Java Application . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19 2.2 Variables and Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

2.2.1 Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23 2.2.2 Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25 2.2.3 Literals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26 2.2.4 Strings and String Literals . . . . . . . . . . . . . . . . . . . . . . . . . . . 27 2.2.5 Variables in Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

2.3 Objects and Subroutines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29 2.3.1 Built-in Subroutines and Functions . . . . . . . . . . . . . . . . . . . . . . 30 2.3.2 Classes and Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33 2.3.3 Operations on Strings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34 2.3.4 Introduction to Enums . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

2.4 Text Input and Output . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37 2.4.1 Basic Output and Formatted Output . . . . . . . . . . . . . . . . . . . . . 38 2.4.2 A First Text Input Example . . . . . . . . . . . . . . . . . . . . . . . . . . 39 2.4.3 Basic TextIO Input Functions . . . . . . . . . . . . . . . . . . . . . . . . . 41 2.4.4 Introduction to File I/O . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43 2.4.5 Other TextIO Features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45 2.4.6 Using Scanner for Input . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

2.5 Details of Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47 2.5.1 Arithmetic Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48 2.5.2 Increment and Decrement . . . . . . . . . . . . . . . . . . . . . . . . . . . 49 2.5.3 Relational Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50 2.5.4 Boolean Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50 2.5.5 Conditional Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

iii

iv *CONTENTS*

2.5.6 Assignment Operators and Type Conversion . . . . . . . . . . . . . . . . 51 2.5.7 Precedence Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53 2.6 Programming Environments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54 2.6.1 Java Development Kit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54 2.6.2 Command Line Environment . . . . . . . . . . . . . . . . . . . . . . . . . 55 2.6.3 Eclipse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57 2.6.4 NetBeans . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60 2.6.5 BlueJ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60 2.6.6 The Problem of Packages . . . . . . . . . . . . . . . . . . . . . . . . . . . 61 Exercises for Chapter 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63 Quiz on Chapter 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

3 Control 67 3.1 Blocks, Loops, and Branches . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67 3.1.1 Blocks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67 3.1.2 The Basic While Loop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68 3.1.3 The Basic If Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71 3.1.4 Definite Assignment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73 3.2 Algorithm Development . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74 3.2.1 Pseudocode and Stepwise Refinement . . . . . . . . . . . . . . . . . . . . 74 3.2.2 The 3N+1 Problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77 3.2.3 Coding, Testing, Debugging . . . . . . . . . . . . . . . . . . . . . . . . . . 80 3.3 while and do..while . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82 3.3.1 The while Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82 3.3.2 The do..while Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84 3.3.3 break and continue . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86 3.4 The for Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88 3.4.1 For Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88 3.4.2 Example: Counting Divisors . . . . . . . . . . . . . . . . . . . . . . . . . . 91 3.4.3 Nested for Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93 3.5 The if Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96 3.5.1 The Dangling else Problem . . . . . . . . . . . . . . . . . . . . . . . . . . 96 3.5.2 Multiway Branching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97 3.5.3 If Statement Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99 3.5.4 The Empty Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103 3.6 The switch Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104 3.6.1 The Basic switch Statement . . . . . . . . . . . . . . . . . . . . . . . . . . 104 3.6.2 Menus and switch Statements . . . . . . . . . . . . . . . . . . . . . . . . . 106 3.6.3 Enums in switch Statements . . . . . . . . . . . . . . . . . . . . . . . . . 108 3.6.4 Definite Assignment and switch Statements . . . . . . . . . . . . . . . . . 108 3.7 Exceptions and try..catch . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109 3.7.1 Exceptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110 3.7.2 try..catch . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110 3.7.3 Exceptions in TextIO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112 3.8 Introduction to Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114 3.8.1 Creating and Using Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . 114 3.8.2 Arrays and For Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116

*CONTENTS* v

3.8.3 Random Access . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117 3.8.4 Partially Full Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119 3.8.5 Two-dimensional Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120

3.9 GUI Programming . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122 3.9.1 Drawing Shapes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122 3.9.2 Drawing in a Program . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125 3.9.3 Animation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

Exercises for Chapter 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129 Quiz on Chapter 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133

4 Subroutines 135 4.1 Black Boxes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135 4.2 Static Subroutines and Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

4.2.1 Subroutine Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137 4.2.2 Calling Subroutines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139 4.2.3 Subroutines in Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140 4.2.4 Member Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143

4.3 Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145 4.3.1 Using Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146 4.3.2 Formal and Actual Parameters . . . . . . . . . . . . . . . . . . . . . . . . 147 4.3.3 Overloading . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148 4.3.4 Subroutine Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149 4.3.5 Array Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151 4.3.6 Command-line Arguments . . . . . . . . . . . . . . . . . . . . . . . . . . . 152 4.3.7 Throwing Exceptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153 4.3.8 Global and Local Variables . . . . . . . . . . . . . . . . . . . . . . . . . . 154

4.4 Return Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155 4.4.1 The return statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155 4.4.2 Function Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 156 4.4.3 3N+1 Revisited . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159

4.5 APIs, Packages, and Javadoc . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160 4.5.1 Toolboxes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160 4.5.2 Java’s Standard Packages . . . . . . . . . . . . . . . . . . . . . . . . . . . 162 4.5.3 Using Classes from Packages . . . . . . . . . . . . . . . . . . . . . . . . . 163 4.5.4 Javadoc . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165 4.5.5 Static Import . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166

4.6 More on Program Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167 4.6.1 Preconditions and Postconditions . . . . . . . . . . . . . . . . . . . . . . . 168 4.6.2 A Design Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 168 4.6.3 The Program . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173

4.7 The Truth About Declarations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175 4.7.1 Initialization in Declarations . . . . . . . . . . . . . . . . . . . . . . . . . 175 4.7.2 Named Constants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 177 4.7.3 Naming and Scope Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . 180

Exercises for Chapter 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 183 Quiz on Chapter 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 186

vi *CONTENTS*

5 Objects and Classes 189 5.1 Objects and Instance Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189 5.1.1 Objects, Classes, and Instances . . . . . . . . . . . . . . . . . . . . . . . . 190 5.1.2 Fundamentals of Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . 192 5.1.3 Getters and Setters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 197 5.1.4 Arrays and Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 198 5.2 Constructors and Object Initialization . . . . . . . . . . . . . . . . . . . . . . . . 199 5.2.1 Initializing Instance Variables . . . . . . . . . . . . . . . . . . . . . . . . . 199 5.2.2 Constructors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 200 5.2.3 Garbage Collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 206 5.3 Programming with Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 207 5.3.1 Some Built-in Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 207 5.3.2 The class “Object” . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 209 5.3.3 Writing and Using a Class . . . . . . . . . . . . . . . . . . . . . . . . . . . 210 5.3.4 Object-oriented Analysis and Design . . . . . . . . . . . . . . . . . . . . . 212 5.4 Programming Example: Card, Hand, Deck . . . . . . . . . . . . . . . . . . . . . . 213 5.4.1 Designing the classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 214 5.4.2 The Card Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 216 5.4.3 Example: A Simple Card Game . . . . . . . . . . . . . . . . . . . . . . . . 220 5.5 Inheritance and Polymorphism . . . . . . . . . . . . . . . . . . . . . . . . . . . . 223 5.5.1 Extending Existing Classes . . . . . . . . . . . . . . . . . . . . . . . . . . 223 5.5.2 Inheritance and Class Hierarchy . . . . . . . . . . . . . . . . . . . . . . . 225 5.5.3 Example: Vehicles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 226 5.5.4 Polymorphism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 229 5.5.5 Abstract Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 231 5.6 this and super . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 234 5.6.1 The Special Variable this . . . . . . . . . . . . . . . . . . . . . . . . . . . 234 5.6.2 The Special Variable super . . . . . . . . . . . . . . . . . . . . . . . . . . 235 5.6.3 super and this As Constructors . . . . . . . . . . . . . . . . . . . . . . . . 236 5.7 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 238 5.7.1 Defining and Implementing Interfaces . . . . . . . . . . . . . . . . . . . . 238 5.7.2 Interfaces as Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 240 5.7.3 Interfaces in Java 8 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 241 5.8 Nested Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 242 5.8.1 Static Nested Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 242 5.8.2 Inner Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 243 5.8.3 Anonymous Inner Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . 244 5.8.4 Java 8 Lambda Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . 245 Exercises for Chapter 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 247 Quiz on Chapter 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 251

6 Introduction to GUI Programming 253 6.1 The Basic GUI Application . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 253 6.1.1 JFrame and JPanel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 255 6.1.2 Components and Layout . . . . . . . . . . . . . . . . . . . . . . . . . . . . 257 6.1.3 Events and Listeners . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 257 6.1.4 Some Java GUI History . . . . . . . . . . . . . . . . . . . . . . . . . . . . 259

*CONTENTS* vii

6.2 Graphics and Painting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 259 6.2.1 Coordinates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 261 6.2.2 Colors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 262 6.2.3 Fonts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 264 6.2.4 Shapes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 264 6.2.5 Graphics2D . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 266 6.2.6 An Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 267 6.2.7 Where is main()? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 269

6.3 Mouse Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 270 6.3.1 Event Handling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 271 6.3.2 MouseEvent and MouseListener . . . . . . . . . . . . . . . . . . . . . . . . 272 6.3.3 MouseEvent Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 275 6.3.4 MouseMotionListeners and Dragging . . . . . . . . . . . . . . . . . . . . . 278 6.3.5 Anonymous Event Handlers . . . . . . . . . . . . . . . . . . . . . . . . . . 281

6.4 Timers, KeyEvents, and State Machines . . . . . . . . . . . . . . . . . . . . . . . 283 6.4.1 Timers and Animation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 283 6.4.2 Keyboard Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 285 6.4.3 Focus Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 288 6.4.4 State Machines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 289

6.5 Basic Components . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 292 6.5.1 JButton . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 294 6.5.2 JLabel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 295 6.5.3 JCheckBox . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 295 6.5.4 JTextField and JTextArea . . . . . . . . . . . . . . . . . . . . . . . . . . . 296 6.5.5 JSlider . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 298

6.6 Basic Layout . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 299 6.6.1 Basic Layout Managers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 300 6.6.2 Borders . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 303 6.6.3 SliderAndButtonDemo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 305 6.6.4 A Simple Calculator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 307 6.6.5 Using a null Layout . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 309 6.6.6 A Little Card Game . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 310

6.7 Menus and Dialogs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 314 6.7.1 Menus and Menubars . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 315 6.7.2 Dialogs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 318 6.7.3 Fine Points of Frames . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 320 6.7.4 Creating Jar Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 321

Exercises for Chapter 6 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 324 Quiz on Chapter 6 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 330

7 Arrays and ArrayLists 331 7.1 Array Details . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 331 7.1.1 For-each Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 332 7.1.2 Variable Arity Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . 333 7.1.3 Array Literals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 335 7.2 Array Processing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 337 7.2.1 Some Processing Examples . . . . . . . . . . . . . . . . . . . . . . . . . . 337

viii *CONTENTS*

7.2.2 Some Standard Array Methods . . . . . . . . . . . . . . . . . . . . . . . . 340 7.2.3 RandomStrings Revisited . . . . . . . . . . . . . . . . . . . . . . . . . . . 341 7.2.4 Dynamic Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 345

7.3 ArrayList . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 347 7.3.1 ArrayList and Parameterized Types . . . . . . . . . . . . . . . . . . . . . 347 7.3.2 Wrapper Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 348 7.3.3 Programming With ArrayList . . . . . . . . . . . . . . . . . . . . . . . . . 350 7.3.4 Vectors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 353

7.4 Searching and Sorting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 354 7.4.1 Searching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 355 7.4.2 Association Lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 357 7.4.3 Insertion Sort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 359 7.4.4 Selection Sort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 361 7.4.5 Unsorting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 363

7.5 Two-dimensional Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 363 7.5.1 The Truth About 2D Arrays . . . . . . . . . . . . . . . . . . . . . . . . . 364 7.5.2 Conway’s Game Of Life . . . . . . . . . . . . . . . . . . . . . . . . . . . . 366 7.5.3 Checkers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 370

Exercises for Chapter 7 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 377 Quiz on Chapter 7 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 381

8 Correctness, Robustness, Efficiency 383 8.1 Introduction to Correctness and Robustness . . . . . . . . . . . . . . . . . . . . . 383 8.1.1 Horror Stories . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 384 8.1.2 Java to the Rescue . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 385 8.1.3 Problems Remain in Java . . . . . . . . . . . . . . . . . . . . . . . . . . . 387 8.2 Writing Correct Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 388 8.2.1 Provably Correct Programs . . . . . . . . . . . . . . . . . . . . . . . . . . 388 8.2.2 Robust Handling of Input . . . . . . . . . . . . . . . . . . . . . . . . . . . 391 8.3 Exceptions and try..catch . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 395 8.3.1 Exceptions and Exception Classes . . . . . . . . . . . . . . . . . . . . . . 396 8.3.2 The try Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 398 8.3.3 Throwing Exceptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 401 8.3.4 Mandatory Exception Handling . . . . . . . . . . . . . . . . . . . . . . . . 403 8.3.5 Programming with Exceptions . . . . . . . . . . . . . . . . . . . . . . . . 404 8.4 Assertions and Annotations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 408 8.4.1 Assertions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 408 8.4.2 Annotations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 411 8.5 Analysis of Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 412 Exercises for Chapter 8 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 418 Quiz on Chapter 8 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 422

9 Linked Data Structures and Recursion 423 9.1 Recursion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 423 9.1.1 Recursive Binary Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . 424 9.1.2 Towers of Hanoi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 426 9.1.3 A Recursive Sorting Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 429 9.1.4 Blob Counting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 431

*CONTENTS* ix

9.2 Linked Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 435 9.2.1 Recursive Linking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 435 9.2.2 Linked Lists . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 437 9.2.3 Basic Linked List Processing . . . . . . . . . . . . . . . . . . . . . . . . . 438 9.2.4 Inserting into a Linked List . . . . . . . . . . . . . . . . . . . . . . . . . . 441 9.2.5 Deleting from a Linked List . . . . . . . . . . . . . . . . . . . . . . . . . . 443

9.3 Stacks, Queues, and ADTs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 444 9.3.1 Stacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 445 9.3.2 Queues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 448 9.3.3 Postfix Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 453

9.4 Binary Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 456 9.4.1 Tree Traversal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 457 9.4.2 Binary Sort Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 459 9.4.3 Expression Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 463

9.5 A Simple Recursive Descent Parser . . . . . . . . . . . . . . . . . . . . . . . . . . 466 9.5.1 Backus-Naur Form . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 467 9.5.2 Recursive Descent Parsing . . . . . . . . . . . . . . . . . . . . . . . . . . . 468 9.5.3 Building an Expression Tree . . . . . . . . . . . . . . . . . . . . . . . . . . 472

Exercises for Chapter 9 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 475 Quiz on Chapter 9 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 478

10 Generic Programming and Collection Classes 481 10.1 Generic Programming . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 481 10.1.1 Generic Programming in Smalltalk . . . . . . . . . . . . . . . . . . . . . . 482 10.1.2 Generic Programming in C++ . . . . . . . . . . . . . . . . . . . . . . . . 483 10.1.3 Generic Programming in Java . . . . . . . . . . . . . . . . . . . . . . . . . 484 10.1.4 The Java Collection Framework . . . . . . . . . . . . . . . . . . . . . . . . 485 10.1.5 Iterators and for-each Loops . . . . . . . . . . . . . . . . . . . . . . . . . . 487 10.1.6 Equality and Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . 489 10.1.7 Generics and Wrapper Classes . . . . . . . . . . . . . . . . . . . . . . . . 491 10.2 Lists and Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 492 10.2.1 ArrayList and LinkedList . . . . . . . . . . . . . . . . . . . . . . . . . . . 492 10.2.2 Sorting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 496 10.2.3 TreeSet and HashSet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 496 10.2.4 EnumSet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 499 10.2.5 Priority Queues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 500 10.3 Maps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 501 10.3.1 The Map Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 502 10.3.2 Views, SubSets, and SubMaps . . . . . . . . . . . . . . . . . . . . . . . . 503 10.3.3 Hash Tables and Hash Codes . . . . . . . . . . . . . . . . . . . . . . . . . 506 10.4 Programming with the JFC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 508 10.4.1 Symbol Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 508 10.4.2 Sets Inside a Map . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 510 10.4.3 Using a Comparator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 513 10.4.4 Word Counting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 514 10.5 Writing Generic Classes and Methods . . . . . . . . . . . . . . . . . . . . . . . . 517 10.5.1 Simple Generic Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 517

x *CONTENTS*

10.5.2 Simple Generic Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . 519 10.5.3 Type Wildcards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 521 10.5.4 Bounded Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 524

Exercises for Chapter 10 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 528 Quiz on Chapter 10 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 532

11 Streams, Files, and Networking 533 11.1 Streams, Readers, and Writers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 533 11.1.1 Character and Byte Streams . . . . . . . . . . . . . . . . . . . . . . . . . 534 11.1.2 PrintWriter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 535 11.1.3 Data Streams . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 537 11.1.4 Reading Text . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 538 11.1.5 The Scanner Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 541 11.1.6 Serialized Object I/O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 542 11.2 Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 543 11.2.1 Reading and Writing Files . . . . . . . . . . . . . . . . . . . . . . . . . . . 543 11.2.2 Files and Directories . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 548 11.2.3 File Dialog Boxes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 550 11.3 Programming With Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 553 11.3.1 Copying a File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 553 11.3.2 Persistent Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 556 11.3.3 Files in GUI Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 558 11.3.4 Storing Objects in Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . 559 11.4 Networking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 567 11.4.1 URLs and URLConnections . . . . . . . . . . . . . . . . . . . . . . . . . . 567 11.4.2 TCP/IP and Client/Server . . . . . . . . . . . . . . . . . . . . . . . . . . 569 11.4.3 Sockets in Java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 571 11.4.4 A Trivial Client/Server . . . . . . . . . . . . . . . . . . . . . . . . . . . . 573 11.4.5 A Simple Network Chat . . . . . . . . . . . . . . . . . . . . . . . . . . . . 577 11.5 A Brief Introduction to XML . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 580 11.5.1 Basic XML Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 581 11.5.2 Working With the DOM . . . . . . . . . . . . . . . . . . . . . . . . . . . . 582 Exercises for Chapter 11 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 588 Quiz on Chapter 11 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 591

12 Threads and Multiprocessing 593 12.1 Introduction to Threads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 593 12.1.1 Creating and Running Threads . . . . . . . . . . . . . . . . . . . . . . . . 594 12.1.2 Operations on Threads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 598 12.1.3 Mutual Exclusion with “synchronized” . . . . . . . . . . . . . . . . . . . . 601 12.1.4 Volatile Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 604 12.2 Programming with Threads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 606 12.2.1 Threads Versus Timers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 606 12.2.2 Recursion in a Thread . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 607 12.2.3 Threads for Background Computation . . . . . . . . . . . . . . . . . . . . 609 12.2.4 Threads for Multiprocessing . . . . . . . . . . . . . . . . . . . . . . . . . . 611 12.2.5 The SwingUtilities Approach . . . . . . . . . . . . . . . . . . . . . . . . . 613 12.3 Threads and Parallel Processing . . . . . . . . . . . . . . . . . . . . . . . . . . . 614

*CONTENTS* xi

12.3.1 Problem Decomposition . . . . . . . . . . . . . . . . . . . . . . . . . . . . 614 12.3.2 Thread Pools and Task Queues . . . . . . . . . . . . . . . . . . . . . . . . 615 12.3.3 Producer/Consumer and Blocking Queues . . . . . . . . . . . . . . . . . . 618 12.3.4 Wait and Notify . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 622

12.4 Threads and Networking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 628 12.4.1 The Blocking I/O Problem . . . . . . . . . . . . . . . . . . . . . . . . . . 628 12.4.2 An Asynchronous Network Chat Program . . . . . . . . . . . . . . . . . . 629 12.4.3 A Threaded Network Server . . . . . . . . . . . . . . . . . . . . . . . . . . 633 12.4.4 Using a Thread Pool . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 635 12.4.5 Distributed Computing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 637

12.5 Network Programming Example . . . . . . . . . . . . . . . . . . . . . . . . . . . 643 12.5.1 The Netgame Framework . . . . . . . . . . . . . . . . . . . . . . . . . . . 644 12.5.2 A Simple Chat Room . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 648 12.5.3 A Networked TicTacToe Game . . . . . . . . . . . . . . . . . . . . . . . . 650 12.5.4 A Networked Poker Game . . . . . . . . . . . . . . . . . . . . . . . . . . . 653

Exercises for Chapter 12 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 654 Quiz on Chapter 12 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 658

13 Advanced GUI Programming 659 13.1 Images and Resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 659 13.1.1 Images and BufferedImages . . . . . . . . . . . . . . . . . . . . . . . . . . 659 13.1.2 Working With Pixels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 666 13.1.3 Resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 668 13.1.4 Cursors and Icons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 669 13.1.5 Image File I/O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 671 13.2 Fancier Graphics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 672 13.2.1 Measuring Text . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 673 13.2.2 Transparency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 675 13.2.3 Antialiasing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 677 13.2.4 Strokes and Paints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 678 13.2.5 Transforms and Shapes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 681 13.3 Actions and Buttons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 684 13.3.1 Action and AbstractAction . . . . . . . . . . . . . . . . . . . . . . . . . . 684 13.3.2 Icons on Buttons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 686 13.3.3 Making Choices . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 687 13.3.4 Toolbars . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 691 13.3.5 Keyboard Accelerators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 692 13.3.6 HTML on Buttons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 694 13.4 Complex Components and MVC . . . . . . . . . . . . . . . . . . . . . . . . . . . 695 13.4.1 Model-View-Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 695 13.4.2 Lists and ListModels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 696 13.4.3 Tables and TableModels . . . . . . . . . . . . . . . . . . . . . . . . . . . . 699 13.4.4 Documents and Editors . . . . . . . . . . . . . . . . . . . . . . . . . . . . 703 13.4.5 Custom Components . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 704 13.5 Finishing Touches . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 709 13.5.1 The Mandelbrot Set . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 709 13.5.2 Design of the Program . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 711

xii *CONTENTS*

13.5.3 Internationalization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 713 13.5.4 Events, Events, Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 715 13.5.5 Custom Dialogs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 717 13.5.6 Preferences . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 718

Exercises for Chapter 13 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 720 Quiz on Chapter 13 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 723

Appendix: Source Files 725 Glossary 735

Preface

Introduction to Programming Using Java is a free introductory computer programming textbook that uses Java as the language of instruction. It is suitable for use in an introductory programming course and for people who are trying to learn programming on their own. There are no prerequisites beyond a general familiarity with the ideas of computers and programs. There is enough material for a full year of college-level programming. Chapters 1 through 7 can be used as a textbook in a one-semester college-level course or in a year-long high school course. The remaining chapters can be covered in a second course.

The Seventh Edition of the book covers “Java 7.” The most recent version of Java is 8, but this book has only a few very short mentions of the new features in Java 8. The home web site for this book is http://math.hws.edu/javanotes/. The page at that address contains links for downloading a copy of the web site and for downloading PDF versions of the book. The web site—and the web site download—includes source code for the sample programs that are discussed in the text, answers to end-of-chapter quizzes and a discussion and solution for each end-of-chapter exercises. Readers are encouraged to download the source code for the examples and to read and run the programs as they read the book. Readers are also strongly encouraged to read the exercise solutions if they want to get the most out of this book. In style, this is a textbook rather than a tutorial. That is, it concentrates on explaining concepts rather than giving step-by-step how-to-do-it guides. I have tried to use a conversa tional writing style that might be closer to classroom lecture than to a typical textbook. This is certainly not a Java reference book, and it is not a comprehensive survey of all the features of Java. It is not written as a quick introduction to Java for people who already know another programming language. Instead, it is directed mainly towards people who are learning program ming for the first time, and it is as much about general programming concepts as it is about Java in particular. I believe that Introduction to Programming using Java is fully competitive with the conventionally published, printed programming textbooks that are available on the market. (Well, all right, I’ll confess that I think it’s better.)

There are several approaches to teaching Java. One approach uses graphical user interface programming from the very beginning. Some people believe that object oriented programming should also be emphasized from the very beginning. This is not the approach that I take. The approach that I favor starts with the more basic building blocks of programming and builds from there. After an introductory chapter, I cover procedural programming in Chapters 2, 3, and 4. Object-oriented programming is introduced in Chapter 5. Chapter 6 covers the closely related topic of event-oriented programming and graphical user interfaces. Arrays are introduced in Chapter 3 with a full treatment in Chapter 7. Chapter 8 is a short chapter that marks a turning point in the book, moving beyond the fundamental ideas of programming to cover more advanced topics. Chapter 8 is about writing robust, correct, and efficient programs. Chapters 9 and 10 cover recursion and data structures, including the Java Collection Framework. Chapter 11 is about files and networking. Chapter 12 covers threads and parallel processing.

xiii

xiv Preface

Finally, Chapter 13 returns to the topic of graphical user interface programming to cover some of Java’s more advanced capabilities.

∗ ∗ ∗

The Seventh Edition of “Introduction to Programming using Java” is not a huge update from the sixth edition. In fact, my main motivation for the new version was to remove any use of applets or coverage of applets from the book. Applets are Java programs that run on a web page. When Java first came out, they were exciting, and it seemed like they would become a major way of creating active content for the Web. Up until the sixth edition, the web pages for this book included applets for running many of the sample programs. However, because of security issues and the emergence of other technologies, applets are no longer widely used. Furthermore, the most recent versions of Java made it fairly difficult and unpleasant to use the applets in the book. In place of applets, I have tried to make it as easy as possible for readers to download the sample programs and run them on their own computers.

Another significant change in the seventh edition is that arrays are now introduced in Chapter 3 in a basic form that is used throughout the next three chapters. Previously, arrays were not introduced until Chapter 7, after objects and GUI programming had already been covered. Much of the more advanced coverage of arrays is still in Chapter 7.

Aside from that, there are many small improvements throughout, mostly related to features that were new in Java 7. Version 7.0.2 is the final release of the seventh edition. ∗ ∗ ∗

The latest complete edition of Introduction to Programming using Java is available on line at http://math.hws.edu/javanotes/. The first version of the book was written in 1996, and there have been several editions since then. All editions are archived (at least until my retirement) at the following Web addresses:

• First edition: http://math.hws.edu/eck/cs124/javanotes1/ (Covers Java 1.0.) • Second edition: http://math.hws.edu/eck/cs124/javanotes2/ (Covers Java 1.1.) • Third edition: http://math.hws.edu/eck/cs124/javanotes3/ (Covers Java 1.1.) • Fourth edition: http://math.hws.edu/eck/cs124/javanotes4/ (Covers Java 1.4.) • Fifth edition: http://math.hws.edu/eck/cs124/javanotes5/ (Covers Java 5.0.) • Sixth edition: http://math.hws.edu/eck/cs124/javanotes6/ (Covers Java 5.0, with a bit of 6.0.)

• Seventh edition: http://math.hws.edu/eck/cs124/javanotes7/ (Covers Java 7.)

Introduction to Programming using Java is free, but it is not in the public domain. Version 7 is published under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc sa/3.0/. For example, you can:

• Post an unmodified copy of the on-line version on your own Web site (including the parts that list the author and state the license under which it is distributed!). • Give away unmodified copies of this book or sell them at cost of production, as long as they meet the requirements of the license.

• Make modified copies of the complete book or parts of it and post them on the web or otherwise distribute them non-commercially, provided that attribution to the author is given, the modifications are clearly noted, and the modified copies are distributed under the same license as the original. This includes translations to other languages.

Preface xv

For uses of the book in ways not covered by the license, permission of the author is required. While it is not actually required by the license, I do appreciate hearing from people who are using or distributing my work.

∗ ∗ ∗

A technical note on production: The on-line and PDF versions of this book are created from a single source, which is written largely in XML. To produce the PDF version, the XML is processed into a form that can be used by the TeX typesetting program. In addition to XML files, the source includes DTDs, XSLT transformations, Java source code files, image files, a TeX macro file, and a couple of scripts that are used in processing. The scripts work on Linux and on Mac OS.

I have made the complete source files available for download at the following address:

http://math.hws.edu/eck/cs124/downloads/javanotes7-full-source.zip

These files were not originally meant for publication, and therefore are not very cleanly written. Furthermore, it requires a fair amount of expertise to use them. However, I have had several requests for the sources and have made them available on an “as-is” basis. For more information about the sources and how they are used see the README file from the source download.

∗ ∗ ∗

Professor David J. Eck

Department of Mathematics and Computer Science

Hobart and William Smith Colleges

300 Pulteney Street

Geneva, New York 14456, USA

Email: eck@hws.edu

WWW: http://math.hws.edu/eck/

xvi Preface

Chapter 1

Overview: The Mental Landscape

When you begin a journey, it’s a good idea to have a mental map of the terrain you’ll be passing through. The same is true for an intellectual journey, such as learning to write computer programs. In this case, you’ll need to know the basics of what computers are and how they work. You’ll want to have some idea of what a computer program is and how one is created. Since you will be writing programs in the Java programming language, you’ll want to know something about that language in particular and about the modern computing environment for which Java is designed.

As you read this chapter, don’t worry if you can’t understand everything in detail. (In fact, it would be impossible for you to learn all the details from the brief expositions in this chapter.) Concentrate on learning enough about the big ideas to orient yourself, in preparation for the rest of the book. Most of what is covered in this chapter will be covered in much greater detail later in the book.

1.1 The Fetch and Execute Cycle: Machine Language

A computer is a complex system consisting of many different components. But at the heart—or the brain, if you want—of the computer is a single component that does the actual computing. This is the Central Processing Unit, or CPU. In a modern desktop computer, the CPU is a single “chip” on the order of one square inch in size. The job of the CPU is to execute programs.

A program is simply a list of unambiguous instructions meant to be followed mechanically by a computer. A computer is built to carry out instructions that are written in a very simple type of language called machine language. Each type of computer has its own machine language, and the computer can directly execute a program only if the program is expressed in that language. (It can execute programs written in other languages if they are first translated into machine language.)

When the CPU executes a program, that program is stored in the computer’s main mem ory (also called the RAM or random access memory). In addition to the program, memory can also hold data that is being used or processed by the program. Main memory consists of a sequence of locations. These locations are numbered, and the sequence number of a location is called its address. An address provides a way of picking out one particular piece of informa tion from among the millions stored in memory. When the CPU needs to access the program instruction or data in a particular location, it sends the address of that information as a sig nal to the memory; the memory responds by sending back the data contained in the specified

1

2 *CHAPTER 1. THE MENTAL LANDSCAPE*

location. The CPU can also store information in memory by specifying the information to be stored and the address of the location where it is to be stored.

On the level of machine language, the operation of the CPU is fairly straightforward (al though it is very complicated in detail). The CPU executes a program that is stored as a sequence of machine language instructions in main memory. It does this by repeatedly reading, or fetching, an instruction from memory and then carrying out, or executing, that instruc tion. This process—fetch an instruction, execute it, fetch another instruction, execute it, and so on forever—is called the fetch-and-execute cycle. With one exception, which will be covered in the next section, this is all that the CPU ever does.

The details of the fetch-and-execute cycle are not terribly important, but there are a few basic things you should know. The CPU contains a few internal registers, which are small memory units capable of holding a single number or machine language instruction. The CPU uses one of these registers—the program counter, or PC—to keep track of where it is in the program it is executing. The PC simply stores the memory address of the next instruction that the CPU should execute. At the beginning of each fetch-and-execute cycle, the CPU checks the PC to see which instruction it should fetch. During the course of the fetch-and-execute cycle, the number in the PC is updated to indicate the instruction that is to be executed in the next cycle. (Usually, but not always, this is just the instruction that sequentially follows the current instruction in the program.)

∗ ∗ ∗

A computer executes machine language programs mechanically—that is without under standing them or thinking about them—simply because of the way it is physically put together. This is not an easy concept. A computer is a machine built of millions of tiny switches called transistors, which have the property that they can be wired together in such a way that an output from one switch can turn another switch on or off. As a computer computes, these switches turn each other on or off in a pattern determined both by the way they are wired together and by the program that the computer is executing.

Machine language instructions are expressed as binary numbers. A binary number is made up of just two possible digits, zero and one. Each zero or one is called a bit. So, a machine language instruction is just a sequence of zeros and ones. Each particular sequence encodes some particular instruction. The data that the computer manipulates is also encoded as binary numbers. In modern computers, each memory location holds a byte, which is a sequence of eight bits. (A machine language instruction or a piece of data generally consists of several bytes, stored in consecutive memory locations.)

A computer can work directly with binary numbers because switches can readily represent such numbers: Turn the switch on to represent a one; turn it off to represent a zero. Machine language instructions are stored in memory as patterns of switches turned on or off. When a machine language instruction is loaded into the CPU, all that happens is that certain switches are turned on or off in the pattern that encodes that instruction. The CPU is built to respond to this pattern by executing the instruction it encodes; it does this simply because of the way all the other switches in the CPU are wired together.

So, you should understand this much about how computers work: Main memory holds machine language programs and data. These are encoded as binary numbers. The CPU fetches machine language instructions from memory one after another and executes them. It does this mechanically, without thinking about or understanding what it does—and therefore the program it executes must be perfect, complete in all details, and unambiguous because the CPU can do nothing but execute it exactly as written. Here is a schematic view of this first-stage

*1.2. ASYNCHRONOUS EVENTS* 3

understanding of the computer:

Memory

10001010 (Location 0)

00110100 (Location 1)

CPU

Program

counter:

0010110111001000

Data to Memory

Data from Memory

Address for

reading/writing

data

01110111 (Location 2) 10100100 (Location 3) 11010010 (Location 4) 10000110 (Location 5) 01001111 (Location 6) 10100000 (Location 7) 00000010 (Location 8) 10100010 (Location 9) 00010100 (Location 10)

...

1.2 Asynchronous Events: Polling Loops and Interrupts

The CPU spends almost all of its time fetching instructions from memory and executing them. However, the CPU and main memory are only two out of many components in a real computer system. A complete system contains other devices such as:

• A hard disk or solid state drive for storing programs and data files. (Note that main memory holds only a comparatively small amount of information, and holds it only as long as the power is turned on. A hard disk or solid state drive is used for permanent storage of larger amounts of information, but programs have to be loaded from there into main memory before they can actually be executed. A hard disk stores data on a spinning magnetic disk, while a solid state drive is a purely electronic device with no moving parts.) • A keyboard and mouse for user input.

• A monitor and printer which can be used to display the computer’s output. • An audio output device that allows the computer to play sounds.

• A network interface that allows the computer to communicate with other computers that are connected to it on a network, either wirelessly or by wire.

• A scanner that converts images into coded binary numbers that can be stored and manipulated on the computer.

The list of devices is entirely open ended, and computer systems are built so that they can easily be expanded by adding new devices. Somehow the CPU has to communicate with and control all these devices. The CPU can only do this by executing machine language instructions (which is all it can do, period). The way this works is that for each device in a system, there is a device driver, which consists of software that the CPU executes when it has to deal with the device. Installing a new device on a system generally has two steps: plugging the device physically into the computer, and installing the device driver software. Without the device driver, the actual physical device would be useless, since the CPU would not be able to communicate with it.

4 *CHAPTER 1. THE MENTAL LANDSCAPE*

∗ ∗ ∗

A computer system consisting of many devices is typically organized by connecting those devices to one or more busses. A bus is a set of wires that carry various sorts of information between the devices connected to those wires. The wires carry data, addresses, and control signals. An address directs the data to a particular device and perhaps to a particular register or location within that device. Control signals can be used, for example, by one device to alert another that data is available for it on the data bus. A fairly simple computer system might be organized like this:

CPU

Input/

Output

Controller

Empty Slot

for future

Memory Disk Drive Expansion

Display Keyboard Network Interface

Data

Address Control

Now, devices such as keyboard, mouse, and network interface can produce input that needs to be processed by the CPU. How does the CPU know that the data is there? One simple idea, which turns out to be not very satisfactory, is for the CPU to keep checking for incoming data over and over. Whenever it finds data, it processes it. This method is called polling, since the CPU polls the input devices continually to see whether they have any input data to report. Unfortunately, although polling is very simple, it is also very inefficient. The CPU can waste an awful lot of time just waiting for input.

To avoid this inefficiency, interrupts are generally used instead of polling. An interrupt is a signal sent by another device to the CPU. The CPU responds to an interrupt signal by putting aside whatever it is doing in order to respond to the interrupt. Once it has handled the interrupt, it returns to what it was doing before the interrupt occurred. For example, when you press a key on your computer keyboard, a keyboard interrupt is sent to the CPU. The CPU responds to this signal by interrupting what it is doing, reading the key that you pressed, processing it, and then returning to the task it was performing before you pressed the key.

Again, you should understand that this is a purely mechanical process: A device signals an interrupt simply by turning on a wire. The CPU is built so that when that wire is turned on, the CPU saves enough information about what it is currently doing so that it can return to the same state later. This information consists of the contents of important internal registers such as the program counter. Then the CPU jumps to some predetermined memory location and begins executing the instructions stored there. Those instructions make up an interrupt handler that does the processing necessary to respond to the interrupt. (This interrupt handler is part of the device driver software for the device that signaled the interrupt.) At the end of the interrupt handler is an instruction that tells the CPU to jump back to what it was doing; it does that by restoring its previously saved state.

*1.2. ASYNCHRONOUS EVENTS* 5

Interrupts allow the CPU to deal with asynchronous events. In the regular fetch-and execute cycle, things happen in a predetermined order; everything that happens is “synchro nized” with everything else. Interrupts make it possible for the CPU to deal efficiently with events that happen “asynchronously,” that is, at unpredictable times.

As another example of how interrupts are used, consider what happens when the CPU needs to access data that is stored on a hard disk. The CPU can access data directly only if it is in main memory. Data on the disk has to be copied into memory before it can be accessed. Unfortunately, on the scale of speed at which the CPU operates, the disk drive is extremely slow. When the CPU needs data from the disk, it sends a signal to the disk drive telling it to locate the data and get it ready. (This signal is sent synchronously, under the control of a regular program.) Then, instead of just waiting the long and unpredictable amount of time that the disk drive will take to do this, the CPU goes on with some other task. When the disk drive has the data ready, it sends an interrupt signal to the CPU. The interrupt handler can then read the requested data.

∗ ∗ ∗

Now, you might have noticed that all this only makes sense if the CPU actually has several tasks to perform. If it has nothing better to do, it might as well spend its time polling for input or waiting for disk drive operations to complete. All modern computers use multitasking to perform several tasks at once. Some computers can be used by several people at once. Since the CPU is so fast, it can quickly switch its attention from one user to another, devoting a fraction of a second to each user in turn. This application of multitasking is called timesharing. But a modern personal computer with just a single user also uses multitasking. For example, the user might be typing a paper while a clock is continuously displaying the time and a file is being downloaded over the network.

Each of the individual tasks that the CPU is working on is called a thread. (Or a process; there are technical differences between threads and processes, but they are not important here, since it is threads that are used in Java.) Many CPUs can literally execute more than one thread simultaneously—such CPUs contain multiple “cores,” each of which can run a thread— but there is always a limit on the number of threads that can be executed at the same time. Since there are often more threads than can be executed simultaneously, the computer has to be able switch its attention from one thread to another, just as a timesharing computer switches its attention from one user to another. In general, a thread that is being executed will continue to run until one of several things happens:

• The thread might voluntarily yield control, to give other threads a chance to run.

• The thread might have to wait for some asynchronous event to occur. For example, the thread might request some data from the disk drive, or it might wait for the user to press a key. While it is waiting, the thread is said to be blocked, and other threads, if any, have a chance to run. When the event occurs, an interrupt will “wake up” the thread so that it can continue running.

• The thread might use up its allotted slice of time and be suspended to allow other threads to run. Not all computers can “forcibly” suspend a thread in this way; those that can are said to use preemptive multitasking. To do preemptive multitasking, a computer needs a special timer device that generates an interrupt at regular intervals, such as 100 times per second. When a timer interrupt occurs, the CPU has a chance to switch from one thread to another, whether the thread that is currently running likes it or not. All modern desktop and laptop computers, and even typical smartphones and tablets, use preemptive

6 *CHAPTER 1. THE MENTAL LANDSCAPE*

multitasking.

Ordinary users, and indeed ordinary programmers, have no need to deal with interrupts and interrupt handlers. They can concentrate on the different tasks or threads that they want the computer to perform; the details of how the computer manages to get all those tasks done are not important to them. In fact, most users, and many programmers, can ignore threads and multitasking altogether. However, threads have become increasingly important as computers have become more powerful and as they have begun to make more use of multitasking and multiprocessing. In fact, the ability to work with threads is fast becoming an essential job skill for programmers. Fortunately, Java has good support for threads, which are built into the Java programming language as a fundamental programming concept. Programming with threads will be covered in Chapter 12.

Just as important in Java and in modern programming in general is the basic concept of asynchronous events. While programmers don’t actually deal with interrupts directly, they do often find themselves writing event handlers, which, like interrupt handlers, are called asyn chronously when specific events occur. Such “event-driven programming” has a very different feel from the more traditional straight-through, synchronous programming. We will begin with the more traditional type of programming, which is still used for programming individual tasks, but we will return to threads and events later in the text, starting in Chapter 6

∗ ∗ ∗

By the way, the software that does all the interrupt handling, handles communication with the user and with hardware devices, and controls which thread is allowed to run is called the operating system. The operating system is the basic, essential software without which a computer would not be able to function. Other programs, such as word processors and Web browsers, are dependent upon the operating system. Common operating systems include Linux, various versions of Windows, and Mac OS.

1.3 The Java Virtual Machine

Machine language consists of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in high-level programming languages such as Java, Fortran, or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called a compiler. A compiler takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language). If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.

There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an interpreter, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so.

*1.3. THE JAVA VIRTUAL MACHINE* 7

One use of interpreters is to execute high-level language programs. For example, the pro gramming language Lisp is usually executed by an interpreter rather than a compiler. However, interpreters have another purpose: they can let you use a machine-language program meant for one type of computer on a completely different type of computer. For example, one of the original home computers was the Commodore 64 or “C64”. While you might not find an actual C64, you can find programs that run on other computers—or even in a web browser—that “emulate” one. Such an emulator can run C64 programs by acting as an interpreter for the C64 machine language.

∗ ∗ ∗

The designers of Java chose to use a combination of compilation and interpreting. Pro grams written in Java are compiled into machine language, but it is a machine language for a computer that doesn’t really exist. This so-called “virtual” computer is known as the Java Virtual Machine, or JVM. The machine language for the Java Virtual Machine is called Java bytecode. There is no reason why Java bytecode couldn’t be used as the machine language of a real computer, rather than a virtual computer. But in fact the use of a virtual machine makes possible one of the main selling points of Java: the fact that it can actually be used on any computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the JVM in the same way that a C64 emulator simulates a Commodore 64 computer. (The term JVM is also used for the Java bytecode interpreter program that does the simulation, so we say that a computer needs a JVM in order to run Java programs. Technically, it would be more correct to say that the interpreter implements the JVM than to say that it is a JVM.)

Of course, a different Java bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program, and the same program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.

Java Interperter

for Mac OS

Java

Java

Compiler Java Interperter

Program

Bytecode Program

for Windows

Java Interperter for Linux

Why, you might wonder, use the intermediate Java bytecode at all? Why not just distribute the original Java program and let each person compile it into the machine language of whatever computer they want to run it on? There are several reasons. First of all, a compiler has to understand Java, a complex high-level language. The compiler is itself a complex program. A Java bytecode interpreter, on the other hand, is a relatively small, simple program. This makes it easy to write a bytecode interpreter for a new type of computer; once that is done, that computer can run any compiled Java program. It would be much harder to write a Java compiler for the same computer.

Furthermore, some Java programs are meant to be downloaded over a network. This leads to obvious security concerns: you don’t want to download and run a program that will damage

8 *CHAPTER 1. THE MENTAL LANDSCAPE*

your computer or your files. The bytecode interpreter acts as a buffer between you and the program you download. You are really running the interpreter, which runs the downloaded program indirectly. The interpreter can protect you from potentially dangerous actions on the part of that program.

When Java was still a new language, it was criticized for being slow: Since Java bytecode was executed by an interpreter, it seemed that Java bytecode programs could never run as quickly as programs compiled into native machine language (that is, the actual machine language of the computer on which the program is running). However, this problem has been largely overcome by the use of just-in-time compilers for executing Java bytecode. A just-in-time compiler translates Java bytecode into native machine language. It does this while it is executing the program. Just as for a normal interpreter, the input to a just-in-time compiler is a Java bytecode program, and its task is to execute that program. But as it is executing the program, it also translates parts of it into machine language. The translated parts of the program can then be executed much more quickly than they could be interpreted. Since a given part of a program is often executed many times as the program runs, a just-in-time compiler can significantly speed up the overall execution time.

I should note that there is no necessary connection between Java and Java bytecode. A program written in Java could certainly be compiled into the machine language of a real com puter. And programs written in other languages can be compiled into Java bytecode. However, the combination of Java and Java bytecode is platform-independent, secure, and network compatible while allowing you to program in a modern high-level object-oriented language.

(In the past few years, it has become fairly common to create new programming languages, or versions of old languages, that compile into Java bytecode. The compiled bytecode programs can then be executed by a standard JVM. New languages that have been developed specifically for programming the JVM include Groovy, Clojure, and Processing. Jython and JRuby are versions of older languages, Python and Ruby, that target the JVM. These languages make it possible to enjoy many of the advantages of the JVM while avoiding some of the technicalities of the Java language. In fact, the use of other languages with the JVM has become important enough that several new features have been added to the JVM specifically to add better support for some of those languages. And this improvement to the JVM has in turn made possible some of the new features in Java 7 and Java 8.)

∗ ∗ ∗

I should also note that the really hard part of platform-independence is providing a “Graph ical User Interface”—with windows, buttons, etc.—that will work on all the platforms that support Java. You’ll see more about this problem in Section 1.6.

1.4 Fundamental Building Blocks of Programs

There are two basic aspects of programming: data and instructions. To work with data, you need to understand variables and types; to work with instructions, you need to understand control structures and subroutines. You’ll spend a large part of the course becoming familiar with these concepts.

A variable is just a memory location (or several consecutive locations treated as a unit) that has been given a name so that it can be easily referred to and used in a program. The programmer only has to worry about the name; it is the compiler’s responsibility to keep track of the memory location. As a programmer, you need to keep in mind that the name refers to

*1.4. BUILDING BLOCKS OF PROGRAMS* 9

a kind of “box” in memory that can hold data, even though you don’t have to know where in memory that box is located.

In Java and in many other programming languages, a variable has a type that indicates what sort of data it can hold. One type of variable might hold integers—whole numbers such as 3, -7, and 0—while another holds floating point numbers—numbers with decimal points such as 3.14, -2.7, or 17.0. (Yes, the computer does make a distinction between the integer 17 and the floating-point number 17.0; they actually look quite different inside the computer.) There could also be types for individual characters (’A’, ’;’, etc.), strings (“Hello”, “A string can include many characters”, etc.), and less common types such as dates, colors, sounds, or any other kind of data that a program might need to store.

Programming languages always have commands for getting data into and out of variables and for doing computations with data. For example, the following “assignment statement,” which might appear in a Java program, tells the computer to take the number stored in the variable named “principal”, multiply that number by 0.07, and then store the result in the variable named “interest”:

interest = principal \* 0.07;

There are also “input commands” for getting data from the user or from files on the computer’s disks, and there are “output commands” for sending data in the other direction. These basic commands—for moving data from place to place and for performing computations—are the building blocks for all programs. These building blocks are combined into complex programs using control structures and subroutines.

∗ ∗ ∗

A program is a sequence of instructions. In the ordinary “flow of control,” the computer executes the instructions in the sequence in which they occur in the program, one after the other. However, this is obviously very limited: the computer would soon run out of instructions to execute. Control structures are special instructions that can change the flow of control. There are two basic types of control structure: loops, which allow a sequence of instructions to be repeated over and over, and branches, which allow the computer to decide between two or more different courses of action by testing conditions that occur as the program is running.

For example, it might be that if the value of the variable “principal” is greater than 10000, then the “interest” should be computed by multiplying the principal by 0.05; if not, then the interest should be computed by multiplying the principal by 0.04. A program needs some way of expressing this type of decision. In Java, it could be expressed using the following “if statement”:

if (principal > 10000)

interest = principal \* 0.05;

else

interest = principal \* 0.04;

(Don’t worry about the details for now. Just remember that the computer can test a condition and decide what to do next on the basis of that test.)

Loops are used when the same task has to be performed more than once. For example, if you want to print out a mailing label for each name on a mailing list, you might say, “Get the first name and address and print the label; get the second name and address and print the label; get the third name and address and print the label. . . ” But this quickly becomes ridiculous—and might not work at all if you don’t know in advance how many names there are. What you would like to say is something like “While there are more names to process, get the

10 *CHAPTER 1. THE MENTAL LANDSCAPE*

next name and address, and print the label.” A loop can be used in a program to express such repetition.

∗ ∗ ∗

Large programs are so complex that it would be almost impossible to write them if there were not some way to break them up into manageable “chunks.” Subroutines provide one way to do this. A subroutine consists of the instructions for performing some task, grouped together as a unit and given a name. That name can then be used as a substitute for the whole set of instructions. For example, suppose that one of the tasks that your program needs to perform is to draw a house on the screen. You can take the necessary instructions, make them into a subroutine, and give that subroutine some appropriate name—say, “drawHouse()”. Then anyplace in your program where you need to draw a house, you can do so with the single command:

drawHouse();

This will have the same effect as repeating all the house-drawing instructions in each place. The advantage here is not just that you save typing. Organizing your program into sub routines also helps you organize your thinking and your program design effort. While writing the house-drawing subroutine, you can concentrate on the problem of drawing a house without worrying for the moment about the rest of the program. And once the subroutine is written, you can forget about the details of drawing houses—that problem is solved, since you have a subroutine to do it for you. A subroutine becomes just like a built-in part of the language which you can use without thinking about the details of what goes on “inside” the subroutine. ∗ ∗ ∗

Variables, types, loops, branches, and subroutines are the basis of what might be called “traditional programming.” However, as programs become larger, additional structure is needed to help deal with their complexity. One of the most effective tools that has been found is object oriented programming, which is discussed in the next section.

1.5 Objects and Object-oriented Programming

Programs must be designed. No one can just sit down at the computer and compose a program of any complexity. The discipline called software engineering is concerned with the construction of correct, working, well-written programs. The software engineer tries to use accepted and proven methods for analyzing the problem to be solved and for designing a program to solve that problem.

During the 1970s and into the 80s, the primary software engineering methodology was structured programming. The structured programming approach to program design was based on the following advice: To solve a large problem, break the problem into several pieces and work on each piece separately; to solve each piece, treat it as a new problem which can itself be broken down into smaller problems; eventually, you will work your way down to problems that can be solved directly, without further decomposition. This approach is called top-down programming.

There is nothing wrong with top-down programming. It is a valuable and often-used ap proach to problem-solving. However, it is incomplete. For one thing, it deals almost entirely with producing the instructions necessary to solve a problem. But as time went on, people realized that the design of the data structures for a program was at least as important as the

*1.5. OBJECT-ORIENTED PROGRAMMING* 11

design of subroutines and control structures. Top-down programming doesn’t give adequate consideration to the data that the program manipulates.

Another problem with strict top-down programming is that it makes it difficult to reuse work done for other projects. By starting with a particular problem and subdividing it into convenient pieces, top-down programming tends to produce a design that is unique to that problem. It is unlikely that you will be able to take a large chunk of programming from another program and fit it into your project, at least not without extensive modification. Producing high-quality programs is difficult and expensive, so programmers and the people who employ them are always eager to reuse past work.

∗ ∗ ∗

So, in practice, top-down design is often combined with bottom-up design. In bottom-up design, the approach is to start “at the bottom,” with problems that you already know how to solve (and for which you might already have a reusable software component at hand). From there, you can work upwards towards a solution to the overall problem.

The reusable components should be as “modular” as possible. A module is a component of a larger system that interacts with the rest of the system in a simple, well-defined, straightforward manner. The idea is that a module can be “plugged into” a system. The details of what goes on inside the module are not important to the system as a whole, as long as the module fulfills its assigned role correctly. This is called information hiding, and it is one of the most important principles of software engineering.

One common format for software modules is to contain some data, along with some sub routines for manipulating that data. For example, a mailing-list module might contain a list of names and addresses along with a subroutine for adding a new name, a subroutine for printing mailing labels, and so forth. In such modules, the data itself is often hidden inside the module; a program that uses the module can then manipulate the data only indirectly, by calling the subroutines provided by the module. This protects the data, since it can only be manipulated in known, well-defined ways. And it makes it easier for programs to use the module, since they don’t have to worry about the details of how the data is represented. Information about the representation of the data is hidden.

Modules that could support this kind of information-hiding became common in program ming languages in the early 1980s. Since then, a more advanced form of the same idea has more or less taken over software engineering. This latest approach is called object-oriented programming, often abbreviated as OOP.

The central concept of object-oriented programming is the object, which is a kind of module containing data and subroutines. The point-of-view in OOP is that an object is a kind of self sufficient entity that has an internal state (the data it contains) and that can respond to messages (calls to its subroutines). A mailing list object, for example, has a state consisting of a list of names and addresses. If you send it a message telling it to add a name, it will respond by modifying its state to reflect the change. If you send it a message telling it to print itself, it will respond by printing out its list of names and addresses.

The OOP approach to software engineering is to start by identifying the objects involved in a problem and the messages that those objects should respond to. The program that results is a collection of objects, each with its own data and its own set of responsibilities. The objects interact by sending messages to each other. There is not much “top-down” in the large-scale design of such a program, and people used to more traditional programs can have a hard time getting used to OOP. However, people who use OOP would claim that object-oriented programs tend to be better models of the way the world itself works, and that they are therefore easier

12 *CHAPTER 1. THE MENTAL LANDSCAPE*

to write, easier to understand, and more likely to be correct.

∗ ∗ ∗

You should think of objects as “knowing” how to respond to certain messages. Different objects might respond to the same message in different ways. For example, a “print” message would produce very different results, depending on the object it is sent to. This property of objects—that different objects can respond to the same message in different ways—is called polymorphism.

It is common for objects to bear a kind of “family resemblance” to one another. Objects that contain the same type of data and that respond to the same messages in the same way belong to the same class. (In actual programming, the class is primary; that is, a class is created and then one or more objects are created using that class as a template.) But objects can be similar without being in exactly the same class.

For example, consider a drawing program that lets the user draw lines, rectangles, ovals, polygons, and curves on the screen. In the program, each visible object on the screen could be represented by a software object in the program. There would be five classes of objects in the program, one for each type of visible object that can be drawn. All the lines would belong to one class, all the rectangles to another class, and so on. These classes are obviously related; all of them represent “drawable objects.” They would, for example, all presumably be able to respond to a “draw yourself” message. Another level of grouping, based on the data needed to represent each type of object, is less obvious, but would be very useful in a program: We can group polygons and curves together as “multipoint objects,” while lines, rectangles, and ovals are “two-point objects.” (A line is determined by its two endpoints, a rectangle by two of its corners, and an oval by two corners of the rectangle that contains it. The rectangles that I am talking about here have sides that are vertical and horizontal, so that they can be specified by just two points; this is the common meaning of “rectangle” in drawing programs.) We could diagram these relationships as follows:

DrawableObject

MultipointObject TwoPointObject

Polygon Curve Line Rectangle Oval

DrawableObject, MultipointObject, and TwoPointObject would be classes in the program. MultipointObject and TwoPointObject would be subclasses of DrawableObject. The class Line would be a subclass of TwoPointObject and (indirectly) of DrawableObject. A subclass of a class is said to inherit the properties of that class. The subclass can add to its inheritance and it can even “override” part of that inheritance (by defining a different response to some method). Nevertheless, lines, rectangles, and so on are drawable objects, and the class DrawableObject expresses this relationship.

Inheritance is a powerful means for organizing a program. It is also related to the problem of reusing software components. A class is the ultimate reusable component. Not only can it

*1.6. THE MODERN USER INTERFACE* 13

be reused directly if it fits exactly into a program you are trying to write, but if it just almost fits, you can still reuse it by defining a subclass and making only the small changes necessary to adapt it exactly to your needs.

So, OOP is meant to be both a superior program-development tool and a partial solution to the software reuse problem. Objects, classes, and object-oriented programming will be important themes throughout the rest of this text. You will start using objects that are built into the Java language in the next chapter, and in Chapter 5 you will begin creating your own classes and objects.

1.6 The Modern User Interface

When computers were first introduced, ordinary people—including most programmers— couldn’t get near them. They were locked up in rooms with white-coated attendants who would take your programs and data, feed them to the computer, and return the computer’s response some time later. When timesharing—where the computer switches its attention rapidly from one person to another—was invented in the 1960s, it became possible for several people to interact directly with the computer at the same time. On a timesharing system, users sit at “terminals” where they type commands to the computer, and the computer types back its re sponse. Early personal computers also used typed commands and responses, except that there was only one person involved at a time. This type of interaction between a user and a computer is called a command-line interface.

Today, of course, most people interact with computers in a completely different way. They use a Graphical User Interface, or GUI. The computer draws interface components on the screen. The components include things like windows, scroll bars, menus, buttons, and icons. Usually, a mouse is used to manipulate such components or, on “touchscreens,” your fingers. Assuming that you have not just been teleported in from the 1970s, you are no doubt already familiar with the basics of graphical user interfaces!

A lot of GUI interface components have become fairly standard. That is, they have similar appearance and behavior on many different computer platforms including Mac OS, Windows, and Linux. Java programs, which are supposed to run on many different platforms without modification to the program, can use all the standard GUI components. They might vary a little in appearance from platform to platform, but their functionality should be identical on any computer on which the program runs.

Shown below is an image of a very simple Java program that demonstrates a few standard GUI interface components. When the program is run, a window similar to the picture shown here will open on the computer screen. There are four components in the window with which the user can interact: a button, a checkbox, a text field, and a pop-up menu. These components are labeled. There are a few other components in the window. The labels themselves are components (even though you can’t interact with them). The right half of the window is a text area component, which can display multiple lines of text. A scrollbar component appears alongside the text area when the number of lines of text becomes larger than will fit in the text area. And in fact, in Java terminology, the whole window is itself considered to be a “component.”

14 *CHAPTER 1. THE MENTAL LANDSCAPE *

(If you would like to run this program, the source code, GUIDemo.java, as well as a compiled program, GUIDemo.jar, are available on line. For more information on using this and other examples from this textbook, see Section 2.6.)

Now, Java actually has two complete sets of GUI components. One of these, the AWT or Abstract Windowing Toolkit, was available in the original version of Java. The other, which is known as Swing, was introduced in Java version 1.2, and is used in preference to the AWT in most modern Java programs. The program that is shown above uses components that are part of Swing.

When a user interacts with GUI components, “events” are generated. For example, clicking a push button generates an event, and pressing return while typing in a text field generates an event. Each time an event is generated, a message is sent to the program telling it that the event has occurred, and the program responds according to its program. In fact, a typical GUI program consists largely of “event handlers” that tell the program how to respond to various types of events. In this example, the program has been programmed to respond to each event by displaying a message in the text area. In a more realistic example, the event handlers would have more to do.

The use of the term “message” here is deliberate. Messages, as you saw in the previous sec tion, are sent to objects. In fact, Java GUI components are implemented as objects. Java includes many predefined classes that represent various types of GUI components. Some of these classes are subclasses of others. Here is a diagram showing just a few of Swing’s GUI classes and their relationships:

JComponent

JLabel JAbstractButton JComboBox JSlider JTextComponent

JButton JToggleButton JCheckBox JRadioButton

JTextField JTextArea

Don’t worry about the details for now, but try to get some feel about how object-oriented programming and inheritance are used here. Note that all the GUI classes are subclasses, directly or indirectly, of a class called JComponent, which represents general properties that are shared by all Swing components. In the diagram, two of the direct subclasses of JComponent themselves have subclasses. The classes JTextArea and JTextField, which have certain behaviors

*1.7. THE INTERNET AND BEYOND* 15

in common, are grouped together as subclasses of JTextComponent. Similarly JButton and JToggleButton are subclasses of JAbstractButton, which represents properties common to both buttons and checkboxes. (JComboBox, by the way, is the Swing class that represents pop-up menus.)

Just from this brief discussion, perhaps you can see how GUI programming can make effec tive use of object-oriented design. In fact, GUIs, with their “visible objects,” are probably a major factor contributing to the popularity of OOP.

Programming with GUI components and events is one of the most interesting aspects of Java. However, we will spend several chapters on the basics before returning to this topic in Chapter 6.

1.7 The Internet and Beyond

Computers can be connected together on networks. A computer on a network can communicate with other computers on the same network by exchanging data and files or by sending and receiving messages. Computers on a network can even work together on a large computation.

Today, millions of computers throughout the world are connected to a single huge network called the Internet. New computers are being connected to the Internet every day, both by wireless communication and by physical connection using technologies such as DSL, cable modems, and Ethernet.

There are elaborate protocols for communication over the Internet. A protocol is simply a detailed specification of how communication is to proceed. For two computers to communicate at all, they must both be using the same protocols. The most basic protocols on the Internet are the Internet Protocol (IP), which specifies how data is to be physically transmitted from one computer to another, and the Transmission Control Protocol (TCP), which ensures that data sent using IP is received in its entirety and without error. These two protocols, which are referred to collectively as TCP/IP, provide a foundation for communication. Other protocols use TCP/IP to send specific types of information such as web pages, electronic mail, and data files.

All communication over the Internet is in the form of packets. A packet consists of some data being sent from one computer to another, along with addressing information that indicates where on the Internet that data is supposed to go. Think of a packet as an envelope with an address on the outside and a message on the inside. (The message is the data.) The packet also includes a “return address,” that is, the address of the sender. A packet can hold only a limited amount of data; longer messages must be divided among several packets, which are then sent individually over the net and reassembled at their destination.

Every computer on the Internet has an IP address, a number that identifies it uniquely among all the computers on the net. (Actually, the claim about uniqueness is not quite true, but the basic idea is valid, and the full truth is complicated.) The IP address is used for addressing packets. A computer can only send data to another computer on the Internet if it knows that computer’s IP address. Since people prefer to use names rather than numbers, most computers are also identified by names, called domain names. For example, the main computer of the Mathematics Department at Hobart and William Smith Colleges has the domain name math.hws.edu. (Domain names are just for convenience; your computer still needs to know IP addresses before it can communicate. There are computers on the Internet whose job it is to translate domain names to IP addresses. When you use a domain name, your computer

16 *CHAPTER 1. THE MENTAL LANDSCAPE*

sends a message to a domain name server to find out the corresponding IP address. Then, your computer uses the IP address, rather than the domain name, to communicate with the other computer.)

The Internet provides a number of services to the computers connected to it (and, of course, to the users of those computers). These services use TCP/IP to send various types of data over the net. Among the most popular services are instant messaging, file sharing, electronic mail, and the World-Wide Web. Each service has its own protocols, which are used to control transmission of data over the network. Each service also has some sort of user interface, which allows the user to view, send, and receive data through the service.

For example, the email service uses a protocol known as SMTP (Simple Mail Transfer Protocol) to transfer email messages from one computer to another. Other protocols, such as POP and IMAP, are used to fetch messages from an email account so that the recipient can read them. A person who uses email, however, doesn’t need to understand or even know about these protocols. Instead, they are used behind the scenes by computer programs to send and receive email messages. These programs provide the user with an easy-to-use user interface to the underlying network protocols.

The World-Wide Web is perhaps the most exciting of network services. The World-Wide Web allows you to request pages of information that are stored on computers all over the Internet. A Web page can contain links to other pages on the same computer from which it was obtained or to other computers anywhere in the world. A computer that stores such pages of information is called a web server. The user interface to the Web is the type of program known as a web browser. Common web browsers include Internet Explorer, Firefox, Chrome, and Safari. You use a Web browser to request a page of information. The browser sends a request for that page to the computer on which the page is stored, and when a response is received from that computer, the web browser displays it to you in a neatly formatted form. A web browser is just a user interface to the Web. Behind the scenes, the web browser uses a protocol called HTTP (HyperText Transfer Protocol) to send each page request and to receive the response from the web server.

∗ ∗ ∗

Now just what, you might be thinking, does all this have to do with Java? In fact, Java is intimately associated with the Internet and the World-Wide Web. When Java was first introduced, one of its big attractions was the ability to write applets. An applet is a small program that is transmitted over the Internet and that runs on a web page. Applets make it possible for a web page to perform complex tasks and have complex interactions with the user. Alas, applets have suffered from a variety of security problems, and fixing those problems has made them more difficult to use. Applets have become much less common on the Web, and in any case, there are other options for running programs on Web pages.

But applets are only one aspect of Java’s relationship with the Internet. Java can be used to write complex, stand-alone applications that do not depend on a Web browser. Many of these programs are network-related. For example many of the largest and most complex web sites use web server software that is written in Java. Java includes excellent support for network protocols, and its platform independence makes it possible to write network programs that work on many different types of computer. You will learn about Java’s network support in Chapter 11.

Its support for networking is not Java’s only advantage. But many good programming languages have been invented only to be soon forgotten. Java has had the good luck to ride on the coattails of the Internet’s immense and increasing popularity.

*1.7. THE INTERNET AND BEYOND* 17

∗ ∗ ∗

As Java has matured, its applications have reached far beyond the Net. The standard version of Java already comes with support for many technologies, such as cryptography and data compression. Free extensions are available to support many other technologies such as advanced sound processing and three-dimensional graphics. Complex, high-performance systems can be developed in Java. For example, Hadoop, a system for large scale data processing, is written in Java. Hadoop is used by Yahoo, Facebook, and other Web sites to process the huge amounts of data generated by their users.

Furthermore, Java is not restricted to use on traditional computers. Java can be used to write programs for many smartphones (though not for the iPhone). It is the primary develop ment language for Android-based devices. (Some mobile devices use a version of Java called Java ME (“Mobile Edition”), but Android uses Google’s own version of Java and does not use the same graphical user interface components as standard Java.) Java is also the programming language for the Amazon Kindle eBook reader and for interactive features on Blu-Ray video disks.

At this time, Java certainly ranks as one of the most widely used programming languages. It is a good choice for almost any programming project that is meant to run on more than one type of computing device, and is a reasonable choice even for many programs that will run on only one device. It is probably still the most widely taught language at Colleges and Universities. It is similar enough to other popular languages, such as C, C++, and Python, that knowing it will give you a good start on learning those languages as well. Overall, learning Java is a great starting point on the road to becoming an expert programmer. I hope you enjoy the journey!

18 *CHAPTER 1. THE MENTAL LANDSCAPE* Quiz on Chapter 1

1. One of the components of a computer is its CPU. What is a CPU and what role does it play in a computer?

2. Explain what is meant by an “asynchronous event.” Give some examples. 3. What is the difference between a “compiler” and an “interpreter”?

4. Explain the difference between high-level languages and machine language.

5. If you have the source code for a Java program, and you want to run that program, you will need both a compiler and an interpreter. What does the Java compiler do, and what does the Java interpreter do?

6. What is a subroutine?

7. Java is an object-oriented programming language. What is an object?

8. What is a variable? (There are four different ideas associated with variables in Java. Try to mention all four aspects in your answer. Hint: One of the aspects is the variable’s name.)

9. Java is a “platform-independent language.” What does this mean?

10. What is the “Internet”? Give some examples of how it is used. (What kind of services does it provide?)

Chapter 2

Programming in the Small I: Names and Things

On a basic level (the level of machine language), a computer can perform only very simple operations. A computer performs complex tasks by stringing together large numbers of such operations. Such tasks must be “scripted” in complete and perfect detail by programs. Creating complex programs will never be really easy, but the difficulty can be handled to some extent by giving the program a clear overall structure. The design of the overall structure of a program is what I call “programming in the large.”

Programming in the small, which is sometimes called coding, would then refer to filling in the details of that design. The details are the explicit, step-by-step instructions for performing fairly small-scale tasks. When you do coding, you are working “close to the machine,” with some of the same concepts that you might use in machine language: memory locations, arithmetic operations, loops and branches. In a high-level language such as Java, you get to work with these concepts on a level several steps above machine language. However, you still have to worry about getting all the details exactly right.

This chapter and the next examine the facilities for programming in the small in the Java programming language. Don’t be misled by the term “programming in the small” into thinking that this material is easy or unimportant. This material is an essential foundation for all types of programming. If you don’t understand it, you can’t write programs, no matter how good you get at designing their large-scale structure.

The last section of this chapter discusses programming environments. That section contains information about how to compile and run Java programs, and you should take a look at it before trying to write and use your own programs or trying to use the sample programs in this book.

2.1 The Basic Java Application

A program is a sequence of instructions that a computer can execute to perform some task. A simple enough idea, but for the computer to make any use of the instructions, they must be written in a form that the computer can use. This means that programs have to be written in programming languages. Programming languages differ from ordinary human languages in being completely unambiguous and very strict about what is and is not allowed in a program. The rules that determine what is allowed are called the syntax of the language. Syntax rules specify the basic vocabulary of the language and how programs can be constructed

19

20 *CHAPTER 2. NAMES AND THINGS*

using things like loops, branches, and subroutines. A syntactically correct program is one that can be successfully compiled or interpreted; programs that have syntax errors will be rejected (hopefully with a useful error message that will help you fix the problem).

So, to be a successful programmer, you have to develop a detailed knowledge of the syntax of the programming language that you are using. However, syntax is only part of the story. It’s not enough to write a program that will run—you want a program that will run and produce the correct result! That is, the meaning of the program has to be right. The meaning of a program is referred to as its semantics. More correctly, the semantics of a programming language is the set of rules that determine the meaning of a program written in that language. A semantically correct program is one that does what you want it to.

Furthermore, a program can be syntactically and semantically correct but still be a pretty bad program. Using the language correctly is not the same as using it well. For example, a good program has “style.” It is written in a way that will make it easy for people to read and to understand. It follows conventions that will be familiar to other programmers. And it has an overall design that will make sense to human readers. The computer is completely oblivious to such things, but to a human reader, they are paramount. These aspects of programming are sometimes referred to as pragmatics. (I will often use the more common term style.)

When I introduce a new language feature, I will explain the syntax, the semantics, and some of the pragmatics of that feature. You should memorize the syntax; that’s the easy part. Then you should get a feeling for the semantics by following the examples given, making sure that you understand how they work, and, ideally, writing short programs of your own to test your understanding. And you should try to appreciate and absorb the pragmatics—this means learning how to use the language feature well, with style that will earn you the admiration of other programmers.

Of course, even when you’ve become familiar with all the individual features of the language, that doesn’t make you a programmer. You still have to learn how to construct complex programs to solve particular problems. For that, you’ll need both experience and taste. You’ll find hints about software development throughout this textbook.

∗ ∗ ∗

We begin our exploration of Java with the problem that has become traditional for such beginnings: to write a program that displays the message “Hello World!”. This might seem like a trivial problem, but getting a computer to do this is really a big first step in learning a new programming language (especially if it’s your first programming language). It means that you understand the basic process of:

1. getting the program text into the computer,

2. compiling the program, and

3. running the compiled program.

The first time through, each of these steps will probably take you a few tries to get right. I won’t go into the details here of how you do each of these steps; it depends on the particular computer and Java programming environment that you are using. See Section 2.6 for informa tion about creating and running Java programs in specific programming environments. But in general, you will type the program using some sort of text editor and save the program in a file. Then, you will use some command to try to compile the file. You’ll either get a message that the program contains syntax errors, or you’ll get a compiled version of the program. In the case of Java, the program is compiled into Java bytecode, not into machine language. Finally, you can run the compiled program by giving some appropriate command. For Java, you will actually use

*2.1. THE BASIC JAVA APPLICATION* 21

an interpreter to execute the Java bytecode. Your programming environment might automate some of the steps for you—for example, the compilation step is often done automatically—but you can be sure that the same three steps are being done in the background.

Here is a Java program to display the message “Hello World!”. Don’t expect to understand what’s going on here just yet; some of it you won’t really understand until a few chapters from now:

/\*\* A program to display the message

\* "Hello World!" on standard output.

\*/

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello World!");

}

} // end of class HelloWorld

The command that actually displays the message is:

System.out.println("Hello World!");

This command is an example of a subroutine call statement. It uses a “built-in subroutine” named System.out.println to do the actual work. Recall that a subroutine consists of the instructions for performing some task, chunked together and given a name. That name can be used to “call” the subroutine whenever that task needs to be performed. A built-in subroutine is one that is already defined as part of the language and therefore automatically available for use in any program.

When you run this program, the message “Hello World!” (without the quotes) will be displayed on standard output. Unfortunately, I can’t say exactly what that means! Java is meant to run on many different platforms, and standard output will mean different things on different platforms. However, you can expect the message to show up in some convenient or inconvenient place. (If you use a command-line interface, like that in Oracle’s Java Development Kit, you type in a command to tell the computer to run the program. The computer will type the output from the program, Hello World!, on the next line. In an integrated development environment such as Eclipse, the output might appear somewhere in one of the environment’s windows.)

You must be curious about all the other stuff in the above program. Part of it consists of comments. Comments in a program are entirely ignored by the computer; they are there for human readers only. This doesn’t mean that they are unimportant. Programs are meant to be read by people as well as by computers, and without comments, a program can be very difficult to understand. Java has two types of comments. The first type begins with // and extends to the end of a line. There is a comment of this form on the last line of the above program. The computer ignores the // and everything that follows it on the same line. The second type of comment starts with /\* and ends with \*/, and it can extend over more than one line. The first three lines of the program are an example of this second type of comment. (A comment that actually begins with /\*\*, like this one does, has special meaning; it is a “Javadoc” comment that can be used to produce documentation for the program.)

Everything else in the program is required by the rules of Java syntax. All programming in Java is done inside “classes.” The first line in the above program (not counting the comment) says that this is a class named HelloWorld. “HelloWorld,” the name of the class, also serves as

22 *CHAPTER 2. NAMES AND THINGS*

the name of the program. Not every class is a program. In order to define a program, a class must include a subroutine named main, with a definition that takes the form:

public static void main(String[] args) {

h*statements* i

}

When you tell the Java interpreter to run the program, the interpreter calls this main() subroutine, and the statements that it contains are executed. These statements make up the script that tells the computer exactly what to do when the program is executed. The main() routine can call other subroutines that are defined in the same class or even in other classes, but it is the main() routine that determines how and in what order the other subroutines are used.

The word “public” in the first line of main() means that this routine can be called from out side the program. This is essential because the main() routine is called by the Java interpreter, which is something external to the program itself. The remainder of the first line of the routine is harder to explain at the moment; for now, just think of it as part of the required syntax. The definition of the subroutine—that is, the instructions that say what it does—consists of the sequence of “statements” enclosed between braces, { and }. Here, I’ve used hstatementsi as a placeholder for the actual statements that make up the program. Throughout this textbook, I will always use a similar format: anything that you see in hthis style of texti (italic in angle brackets) is a placeholder that describes something you need to type when you write an actual program.

As noted above, a subroutine can’t exist by itself. It has to be part of a “class”. A program is defined by a public class that takes the form:

public class h*program-name* i {

h*optional-variable-declarations-and-subroutines* i

public static void main(String[] args) {

h*statements* i

}

h*optional-variable-declarations-and-subroutines* i

}

The name on the first line is the name of the program, as well as the name of the class. (Remember, again, that hprogram-namei is a placeholder for the actual name!) If the name of the class is HelloWorld, then the class must be saved in a file called HelloWorld.java. When this file is compiled, another file named HelloWorld.class will be produced. This class file, HelloWorld.class, contains the translation of the program into Java bytecode, which can be executed by a Java interpreter. HelloWorld.java is called the source code for the program. To execute the program, you only need the compiled class file, not the source code.

The layout of the program on the page, such as the use of blank lines and indentation, is not part of the syntax or semantics of the language. The computer doesn’t care about layout—you could run the entire program together on one line as far as it is concerned. However, layout is important to human readers, and there are certain style guidelines for layout that are followed by most programmers.

*2.2. VARIABLES AND TYPES* 23

Also note that according to the above syntax specification, a program can contain other subroutines besides main(), as well as things called “variable declarations.” You’ll learn more about these later, but not until Chapter 4.

2.2 Variables and the Primitive Types

Names are fundamental to programming. In programs, names are used to refer to many different sorts of things. In order to use those things, a programmer must understand the rules for giving names to them and the rules for using the names to work with them. That is, the programmer must understand the syntax and the semantics of names.

According to the syntax rules of Java, the most basic names are identifiers. Identifiers can be used to name classes, variables, and subroutines. An identifier is a sequence of one or more characters. It must begin with a letter or underscore and must consist entirely of letters, digits, and underscores. (“Underscore” refers to the character ’ ’.) For example, here are some legal identifiers:

N n rate x15 quite a long name HelloWorld

No spaces are allowed in identifiers; HelloWorld is a legal identifier, but “Hello World” is not. Upper case and lower case letters are considered to be different, so that HelloWorld, helloworld, HELLOWORLD, and hElloWorLD are all distinct names. Certain words are reserved for special uses in Java, and cannot be used as identifiers. These reserved words include: class, public, static, if, else, while, and several dozen other words. (Remember that reserved words are not identifiers, since they can’t be used as names for things.)

Java is actually pretty liberal about what counts as a letter or a digit. Java uses the Unicode character set, which includes thousands of characters from many different languages and different alphabets, and many of these characters count as letters or digits. However, I will be sticking to what can be typed on a regular English keyboard.

The pragmatics of naming includes style guidelines about how to choose names for things. For example, it is customary for names of classes to begin with upper case letters, while names of variables and of subroutines begin with lower case letters; you can avoid a lot of confusion by following this standard convention in your own programs. Most Java programmers do not use underscores in names, although some do use them at the beginning of the names of certain kinds of variables. When a name is made up of several words, such as HelloWorld or interestRate, it is customary to capitalize each word, except possibly the first; this is sometimes referred to as camel case, since the upper case letters in the middle of a name are supposed to look something like the humps on a camel’s back.

Finally, I’ll note that in addition to simple identifiers, things in Java can have compound names which consist of several simple names separated by periods. (Compound names are also called qualified names.) You’ve already seen an example: System.out.println. The idea here is that things in Java can contain other things. A compound name is a kind of path to an item through one or more levels of containment. The name System.out.println indicates that something called “System” contains something called “out” which in turn contains something called “println”.

2.2.1 Variables

Programs manipulate data that are stored in memory. In machine language, data can only be referred to by giving the numerical address of the location in memory where the data is stored.

24 *CHAPTER 2. NAMES AND THINGS*

In a high-level language such as Java, names are used instead of numbers to refer to data. It is the job of the computer to keep track of where in memory the data is actually stored; the programmer only has to remember the name. A name used in this way—to refer to data stored in memory—is called a variable.

Variables are actually rather subtle. Properly speaking, a variable is not a name for the data itself but for a location in memory that can hold data. You should think of a variable as a container or box where you can store data that you will need to use later. The variable refers directly to the box and only indirectly to the data in the box. Since the data in the box can change, a variable can refer to different data values at different times during the execution of the program, but it always refers to the same box. Confusion can arise, especially for beginning programmers, because when a variable is used in a program in certain ways, it refers to the container, but when it is used in other ways, it refers to the data in the container. You’ll see examples of both cases below.

(In this way, a variable is something like the title, “The President of the United States.” This title can refer to different people at different times, but it always refers to the same office. If I say “the President is playing basketball,” I mean that Barack Obama is playing basketball. But if I say “Hillary Clinton wants to be President” I mean that she wants to fill the office, not that she wants to be Barack Obama.)

In Java, the only way to get data into a variable—that is, into the box that the variable names—is with an assignment statement. An assignment statement takes the form:

h*variable* i = h*expression* i;

where hexpressioni represents anything that refers to or computes a data value. When the computer comes to an assignment statement in the course of executing a program, it evaluates the expression and puts the resulting data value into the variable. For example, consider the simple assignment statement

rate = 0.07;

The hvariablei in this assignment statement is rate, and the hexpressioni is the number 0.07. The computer executes this assignment statement by putting the number 0.07 in the variable rate, replacing whatever was there before. Now, consider the following more complicated assignment statement, which might come later in the same program:

interest = rate \* principal;

Here, the value of the expression “rate \* principal” is being assigned to the variable interest. In the expression, the \* is a “multiplication operator” that tells the computer to multiply rate times principal. The names rate and principal are themselves variables, and it is really the values stored in those variables that are to be multiplied. We see that when a variable is used in an expression, it is the value stored in the variable that matters; in this case, the variable seems to refer to the data in the box, rather than to the box itself. When the computer executes this assignment statement, it takes the value of rate, multiplies it by the value of principal, and stores the answer in the box referred to by interest. When a variable is used on the left-hand side of an assignment statement, it refers to the box that is named by the variable.

(Note, by the way, that an assignment statement is a command that is executed by the computer at a certain time. It is not a statement of fact. For example, suppose a program includes the statement “rate = 0.07;”. If the statement “interest = rate \* principal;” is executed later in the program, can we say that the principal is multiplied by 0.07? No!

*2.2. VARIABLES AND TYPES* 25

The value of rate might have been changed in the meantime by another statement. The meaning of an assignment statement is completely different from the meaning of an equation in mathematics, even though both use the symbol “=”.)

2.2.2 Types

A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if you try to violate this rule by assigning a variable of the wrong type to a variable. We say that Java is a strongly typed language because it enforces this rule.

There are eight so-called primitive types built into Java. The primitive types are named byte, short, int, long, float, double, char, and boolean. The first four types hold integers (whole numbers such as 17, -38477, and 0). The four integer types are distinguished by the ranges of integers they can hold. The float and double types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type char holds a single character from the Unicode character set. And a variable of type boolean holds one of the two logical values true or false.

Any data value stored in the computer’s memory must be represented as a binary number, that is as a string of zeros and ones. A single zero or one is called a bit. A string of eight bits is called a byte. Memory is usually measured in terms of bytes. Not surprisingly, the byte data type refers to a single byte of memory. A variable of type byte holds a string of eight bits, which can represent any of the integers between -128 and 127, inclusive. (There are 256 integers in that range; eight bits can represent 256—two raised to the power eight—different values.) As for the other integer types,

• short corresponds to two bytes (16 bits). Variables of type short have values in the range -32768 to 32767.

• int corresponds to four bytes (32 bits). Variables of type int have values in the range -2147483648 to 2147483647.

• long corresponds to eight bytes (64 bits). Variables of type long have values in the range -9223372036854775808 to 9223372036854775807.

You don’t have to remember these numbers, but they do give you some idea of the size of integers that you can work with. Usually, for representing integer data you should just stick to the int data type, which is good enough for most purposes.

The float data type is represented in four bytes of memory, using a standard method for encoding real numbers. The maximum value for a float is about 10 raised to the power 38. A float can have about 7 significant digits. (So that 32.3989231134 and 32.3989234399 would both have to be rounded off to about 32.398923 in order to be stored in a variable of type float.) A double takes up 8 bytes, can range up to about 10 to the power 308, and has about 15 significant digits. Ordinarily, you should stick to the double type for real values.

A variable of type char occupies two bytes in memory. The value of a char variable is a single character such as A, \*, x, or a space character. The value can also be a special character such a tab or a carriage return or one of the many Unicode characters that come from different languages. Values of type char are closely related to integer values, since a character is actually stored as a 16-bit integer code number. In fact, we will see that chars in Java can actually be used like integers in certain situations.

It is important to remember that a primitive type value is represented using ony a certain, finite number of bits. So, an int can’t be an arbitrary integer; it can only be an integer

26 *CHAPTER 2. NAMES AND THINGS*

in a certain finite range of values. Similarly, float and double variables can only take on certain values. They are not true real numbers in the mathematical sense. For example, the mathematical constant π can only be approximated by a value of type float or double, since it would require an infinite number of decimal places to represent it exactly. For that matter, simple numbers like 1/3 can only be approximated by floats and doubles.

2.2.3 Literals

A data value is stored in the computer as a sequence of bits. In the computer’s memory, it doesn’t look anything like a value written on this page. You need a way to include constant values in the programs that you write. In a program, you represent constant values as literals. A literal is something that you can type in a program to represent a value. It is a kind of name for a constant value.

For example, to type a value of type char in a program, you must surround it with a pair of single quote marks, such as ’A’, ’\*’, or ’x’. The character and the quote marks make up a literal of type char. Without the quotes, A would be an identifier and \* would be a multiplication operator. The quotes are not part of the value and are not stored in the variable; they are just a convention for naming a particular character constant in a program. If you want to store the character A in a variable ch of type char, you could do so with the assignment statement

ch = ’A’;

Certain special characters have special literals that use a backslash, \, as an “escape character”. In particular, a tab is represented as ’\t’, a carriage return as ’\r’, a linefeed as ’\n’, the single quote character as ’\’’, and the backslash itself as ’\\’. Note that even though you type two characters between the quotes in ’\t’, the value represented by this literal is a single tab character.

Numeric literals are a little more complicated than you might expect. Of course, there are the obvious literals such as 317 and 17.42. But there are other possibilities for expressing numbers in a Java program. First of all, real numbers can be represented in an exponential form such as 1.3e12 or 12.3737e-108. The “e12” and “e-108” represent powers of 10, so that 1.3e12 means 1.3 times 1012 and 12.3737e-108 means 12.3737 times 10−108. This format can be used to express very large and very small numbers. Any numeric literal that contains a decimal point or exponential is a literal of type double. To make a literal of type float, you have to append an “F” or “f” to the end of the number. For example, “1.2F” stands for 1.2 considered as a value of type float. (Occasionally, you need to know this because the rules of Java say that you can’t assign a value of type double to a variable of type float, so you might be confronted with a ridiculous-seeming error message if you try to do something like “x = 1.2;” if x is a variable of type float. You have to say “x = 1.2F;". This is one reason why I advise sticking to type double for real numbers.)

Even for integer literals, there are some complications. Ordinary integers such as 177777 and -32 are literals of type byte, short, or int, depending on their size. You can make a literal of type long by adding “L” as a suffix. For example: 17L or 728476874368L. As another complication, Java allows binary, octal (base-8), and hexadecimal (base-16) literals. I don’t want to cover number bases in detail, but in case you run into them in other people’s programs, it’s worth knowing a few things: Octal numbers use only the digits 0 through 7. In Java, a numeric literal that begins with a 0 is interpreted as an octal number; for example, the octal literal 045 represents the number 37, not the number 45. Octal numbers are rarely used, but you need to be aware of what happens when you start a number with a zero. Hexadecimal

*2.2. VARIABLES AND TYPES* 27

numbers use 16 digits, the usual digits 0 through 9 and the letters A, B, C, D, E, and F. Upper case and lower case letters can be used interchangeably in this context. The letters represent the numbers 10 through 15. In Java, a hexadecimal literal begins with 0x or 0X, as in 0x45 or 0xFF7A. Finally, binary literals start with 0b or 0B and contain only the digits 0 and 1; for example: 0b10110.

As a final complication, numeric literals in Java 7 can include the underscore character (“ ”), which can be used to separate groups of digits. For example, the integer constant for seven billion could be written 7 000 000 000, which is a good deal easier to decipher than 7000000000. There is no rule about how many digits have to be in each group. Underscores can be especially useful in long binary numbers; for example, 0b1010 1100 1011.

I will note that hexadecimal numbers can also be used in character literals to represent arbitrary Unicode characters. A Unicode literal consists of \u followed by four hexadecimal digits. For example, the character literal ’\u00E9’ represents the Unicode character that is an “e” with an acute accent.

For the type boolean, there are precisely two literals: true and false. These literals are typed just as I’ve written them here, without quotes, but they represent values, not variables. Boolean values occur most often as the values of conditional expressions. For example,

rate > 0.05

is a boolean-valued expression that evaluates to true if the value of the variable rate is greater than 0.05, and to false if the value of rate is not greater than 0.05. As you’ll see in Chapter 3, boolean-valued expressions are used extensively in control structures. Of course, boolean values can also be assigned to variables of type boolean. For example, if test is a variable of type boolean, then both of the following assignment statements are legal:

test = true;

test = rate > 0.05;

2.2.4 Strings and String Literals

Java has other types in addition to the primitive types, but all the other types represent objects rather than “primitive” data values. For the most part, we are not concerned with objects for the time being. However, there is one predefined object type that is very important: the type String. (String is a type, but not a primitive type; it is in fact the name of a class, and we will return to that aspect of strings in the next section.)

A value of type String is a sequence of characters. You’ve already seen a string literal: "Hello World!". The double quotes are part of the literal; they have to be typed in the program. However, they are not part of the actual String value, which consists of just the characters between the quotes. A string can contain any number of characters, even zero. A string with no characters is called the empty string and is represented by the literal "", a pair of double quote marks with nothing between them. Remember the difference between single quotes and double quotes! Single quotes are used for char literals and double quotes for String literals! There is a big difference between the String "A" and the char ’A’.

Within a string literal, special characters can be represented using the backslash notation. Within this context, the double quote is itself a special character. For example, to represent the string value

I said, "Are you listening!"

with a linefeed at the end, you would have to type the string literal:

28 *CHAPTER 2. NAMES AND THINGS*

"I said, \"Are you listening!\"\n"

You can also use \t, \r, \\, and Unicode sequences such as \u00E9 to represent other special characters in string literals.

2.2.5 Variables in Programs

A variable can be used in a program only if it has first been declared. A variable declaration statement is used to declare one or more variables and to give them names. When the computer executes a variable declaration, it sets aside memory for the variable and associates the variable’s name with that memory. A simple variable declaration takes the form:

h*type-name* i h*variable-name-or-names* i;

The hvariable-name-or-namesi can be a single variable name or a list of variable names separated by commas. (We’ll see later that variable declaration statements can actually be somewhat more complicated than this.) Good programming style is to declare only one variable in a declaration statement, unless the variables are closely related in some way. For example:

int numberOfStudents;

String name;

double x, y;

boolean isFinished;

char firstInitial, middleInitial, lastInitial;

It is also good style to include a comment with each variable declaration to explain its purpose in the program, or to give other information that might be useful to a human reader. For example:

double principal; // Amount of money invested.

double interestRate; // Rate as a decimal, not percentage.

In this chapter, we will only use variables declared inside the main() subroutine of a pro gram. Variables declared inside a subroutine are called local variables for that subroutine. They exist only inside the subroutine, while it is running, and are completely inaccessible from outside. Variable declarations can occur anywhere inside the subroutine, as long as each vari able is declared before it is used in any way. Some people like to declare all the variables at the beginning of the subroutine. Others like to wait to declare a variable until it is needed. My preference: Declare important variables at the beginning of the subroutine, and use a comment to explain the purpose of each variable. Declare “utility variables” which are not important to the overall logic of the subroutine at the point in the subroutine where they are first used. Here is a simple program using some variables and assignment statements:

/\*\*

\* This class implements a simple program that

\* will compute the amount of interest that is

\* earned on $17,000 invested at an interest

\* rate of 0.027 for one year. The interest and

\* the value of the investment after one year are

\* printed to standard output.

\*/

public class Interest {

public static void main(String[] args) {

*2.3. OBJECTS AND SUBROUTINES* 29

/\* Declare the variables. \*/

double principal; // The value of the investment.

double rate; // The annual interest rate.

double interest; // Interest earned in one year.

/\* Do the computations. \*/

principal = 17000;

rate = 0.027;

interest = principal \* rate; // Compute the interest.

principal = principal + interest;

// Compute value of investment after one year, with interest.

// (Note: The new value replaces the old value of principal.)

/\* Output the results. \*/

System.out.print("The interest earned is $");

System.out.println(interest);

System.out.print("The value of the investment after one year is $"); System.out.println(principal);

} // end of main()

} // end of class Interest

This program uses several subroutine call statements to display information to the user of the program. Two different subroutines are used: System.out.print and System.out.println. The difference between these is that System.out.println adds a linefeed after the end of the information that it displays, while System.out.print does not. Thus, the value of interest, which is displayed by the subroutine call “System.out.println(interest);”, follows on the same line as the string displayed by the previous System.out.print statement. Note that the value to be displayed by System.out.print or System.out.println is provided in parentheses after the subroutine name. This value is called a parameter to the subroutine. A parameter provides a subroutine with information it needs to perform its task. In a subroutine call state ment, any parameters are listed in parentheses after the subroutine name. Not all subroutines have parameters. If there are no parameters in a subroutine call statement, the subroutine name must be followed by an empty pair of parentheses.

All the sample programs for this textbook are available in separate source code files in the on-line version of this text at http://math.hws.edu/javanotes/source. They are also included in the downloadable archives of the web site, in a folder named source. The source code for the Interest program, for example, can be found in the file Interest.java in subfolder named chapter2 inside the source folder.

2.3 Strings, Classes, Objects, and Subroutines

The previous section introduced the eight primitive data types and the type String. There is a fundamental difference between the primitive types and String: Values of type String are objects. While we will not study objects in detail until Chapter 5, it will be useful for you to know a little about them and about a closely related topic: classes. This is not just because strings are useful but because objects and classes are essential to understanding another important programming concept, subroutines.

30 *CHAPTER 2. NAMES AND THINGS*

2.3.1 Built-in Subroutines and Functions

Recall that a subroutine is a set of program instructions that have been chunked together and given a name. A subroutine is designed to perform some task. To get that task performed in a program, you can “call” the subroutine using a subroutine call statement. In Chapter 4, you’ll learn how to write your own subroutines, but you can get a lot done in a program just by calling subroutines that have already been written for you. In Java, every subroutine is contained either in a class or in an object. Some classes that are standard parts of the Java language contain predefined subroutines that you can use. A value of type String, which is an object, contains subroutines that can be used to manipulate that string. These subroutines are “built into” the Java language. You can call all these subroutines without understanding how they were written or how they work. Indeed, that’s the whole point of subroutines: A subroutine is a “black box” which can be used without knowing what goes on inside.

Let’s first consider subroutines that are part of a class. One of the purposes of a class is to group together some variables and subroutines, which are contained in that class. These variables and subroutines are called static members of the class. You’ve seen one example: In a class that defines a program, the main() routine is a static member of the class. The parts of a class definition that define static members are marked with the reserved word “static”, such as the word “static” in public static void main...

When a class contains a static variable or subroutine, the name of the class is part of the full name of the variable or subroutine. For example, the standard class named System contains a subroutine named exit. To use that subroutine in your program, you must refer to it as System.exit. This full name consists of the name of the class that contains the subroutine, followed by a period, followed by the name of the subroutine. This subroutine requires an integer as parameter, so you would actually use it with a subroutine call statement such as

System.exit(0);

Calling System.exit will terminate the program and shut down the Java Virtual Machine. You could use it if you had some reason to terminate the program before the end of the main routine. (The parameter tells the computer why the program was terminated. A parameter value of 0 indicates that the program ended normally. Any other value indicates that the program was terminated because an error was detected, so you could call System.exit(1) to indicate that the program is ending because of an error. The parameter is sent back to the operating system; in practice, the value is usually ignored by the operating system.)

System is just one of many standard classes that come with Java. Another useful class is called Math. This class gives us an example of a class that contains static variables: It includes the variables Math.PI and Math.E whose values are the mathematical constants π and e. Math also contains a large number of mathematical “functions.” Every subroutine performs some specific task. For some subroutines, that task is to compute or retrieve some data value. Subroutines of this type are called functions. We say that a function returns a value. Generally, the returned value is meant to be used somehow in the program that calls the function.

You are familiar with the mathematical function that computes the square root of a number. The corresponding function in Java is called Math.sqrt. This function is a static member subroutine of the class named Math. If x is any numerical value, then Math.sqrt(x) computes and returns the square root of that value. Since Math.sqrt(x) represents a value, it doesn’t make sense to put it on a line by itself in a subroutine call statement such as

Math.sqrt(x); // This doesn’t make sense!

*2.3. OBJECTS AND SUBROUTINES* 31

What, after all, would the computer do with the value computed by the function in this case? You have to tell the computer to do something with the value. You might tell the computer to display it:

System.out.print( Math.sqrt(x) ); // Display the square root of x. or you might use an assignment statement to tell the computer to store that value in a variable: lengthOfSide = Math.sqrt(x);

The function call Math.sqrt(x) represents a value of type double, and it can be used anyplace where a numeric literal of type double could be used.

The Math class contains many static member functions. Here is a list of some of the more important of them:

• Math.abs(x), which computes the absolute value of x.

• The usual trigonometric functions, Math.sin(x), Math.cos(x), and Math.tan(x). (For all the trigonometric functions, angles are measured in radians, not degrees.)

• The inverse trigonometric functions arcsin, arccos, and arctan, which are written as: Math.asin(x), Math.acos(x), and Math.atan(x). The return value is expressed in radi ans, not degrees.

• The exponential function Math.exp(x) for computing the number e raised to the power x, and the natural logarithm function Math.log(x) for computing the logarithm of x in the base e.

• Math.pow(x,y) for computing x raised to the power y.

• Math.floor(x), which rounds x down to the nearest integer value that is less than or equal to x. Even though the return value is mathematically an integer, it is returned as a value of type double, rather than of type int as you might expect. For example, Math.floor(3.76) is 3.0. The function Math.round(x) returns the integer that is closest to x, and Math.ceil(x) rounds x up to an integer. (“Ceil” is short for “ceiling”, the opposite of “floor.”)

• Math.random(), which returns a randomly chosen double in the range 0.0 <= Math.random() < 1.0. (The computer actually calculates so-called “pseudorandom” numbers, which are not truly random but are effectively random enough for most pur poses.) We will find a lot of uses for Math.random in future examples.

For these functions, the type of the parameter—the x or y inside the parentheses—can be any value of any numeric type. For most of the functions, the value returned by the function is of type double no matter what the type of the parameter. However, for Math.abs(x), the value returned will be the same type as x; if x is of type int, then so is Math.abs(x). So, for example, while Math.sqrt(9) is the double value 3.0, Math.abs(9) is the int value 9.

Note that Math.random() does not have any parameter. You still need the parentheses, even though there’s nothing between them. The parentheses let the computer know that this is a subroutine rather than a variable. Another example of a subroutine that has no parameters is the function System.currentTimeMillis(), from the System class. When this function is executed, it retrieves the current time, expressed as the number of milliseconds that have passed since a standardized base time (the start of the year 1970, if you care). One millisecond is one thousandth of a second. The return value of System.currentTimeMillis() is of type long (a 64-bit integer). This function can be used to measure the time that it takes the computer to

32 *CHAPTER 2. NAMES AND THINGS*

perform a task. Just record the time at which the task is begun and the time at which it is finished and take the difference.

Here is a sample program that performs a few mathematical tasks and reports the time that it takes for the program to run. On some computers, the time reported might be zero, because it is too small to measure in milliseconds. Even if it’s not zero, you can be sure that most of the time reported by the computer was spent doing output or working on tasks other than the program, since the calculations performed in this program occupy only a tiny fraction of a millisecond of a computer’s time.

/\*\*

\* This program performs some mathematical computations and displays the \* results. It also displays the value of the constant Math.PI. It then \* reports the number of seconds that the computer spent on this task. \*/

public class TimedComputation {

public static void main(String[] args) {

long startTime; // Starting time of program, in milliseconds.

long endTime; // Time when computations are done, in milliseconds. double time; // Time difference, in seconds.

startTime = System.currentTimeMillis();

double width, height, hypotenuse; // sides of a triangle

width = 42.0;

height = 17.0;

hypotenuse = Math.sqrt( width\*width + height\*height );

System.out.print("A triangle with sides 42 and 17 has hypotenuse "); System.out.println(hypotenuse);

System.out.println("\nMathematically, sin(x)\*sin(x) + "

+ "cos(x)\*cos(x) - 1 should be 0.");

System.out.println("Let’s check this for x = 1:");

System.out.print(" sin(1)\*sin(1) + cos(1)\*cos(1) - 1 is ");

System.out.println( Math.sin(1)\*Math.sin(1)

+ Math.cos(1)\*Math.cos(1) - 1 );

System.out.println("(There can be round-off errors when"

+ " computing with real numbers!)");

System.out.print("\nHere is a random number: ");

System.out.println( Math.random() );

System.out.print("The value of Math.PI is ");

System.out.println( Math.PI );

endTime = System.currentTimeMillis();

time = (endTime - startTime) / 1000.0;

System.out.print("\nRun time in seconds was: ");

System.out.println(time);

} // end main()

} // end class TimedComputation

*2.3. OBJECTS AND SUBROUTINES* 33 2.3.2 Classes and Objects

Classes can be containers for static variables and subroutines. However classes also have another purpose. They are used to describe objects. In this role, the class is a type, in the same way that int and double are types. That is, the class name can be used to declare variables. Such variables can only hold one type of value. The values in this case are objects. An object is a collection of variables and subroutines. Every object has an associated class that tells what “type” of object it is. The class of an object specifies what subroutines and variables that object contains. All objects defined by the same class are similar in that they contain similar collections of variables and subroutines. For example, an object might represent a point in the plane, and it might contain variables named x and y to represent the coordinates of that point. Every point object would have an x and a y, but different points would have different values for these variables. A class, named Point, for example, could exist to define the common structure of all point objects, and all such objects would then be values of type Point.

As another example, let’s look again at System.out.println. System is a class, and out is a static variable within that class. However, the value of System.out is an object, and System.out.println is actually the full name of a subroutine that is contained in the ob ject System.out. You don’t need to understand it at this point, but the object referred to by System.out is an object of the class PrintStream. PrintStream is another class that is a standard part of Java. Any object of type PrintStream is a destination to which information can be printed; any object of type PrintStream has a println subroutine that can be used to send information to that destination. The object System.out is just one possible desti nation, and System.out.println is a subroutine that sends information to that particular destination. Other objects of type PrintStream might send information to other destinations such as files or across a network to other computers. This is object-oriented programming: Many different things which have something in common—they can all be used as destina tions for information—can all be used in the same way—through a println subroutine. The PrintStream class expresses the commonalities among all these objects.

The dual role of classes can be confusing, and in practice most classes are designed to perform primarily or exclusively in only one of the two possible roles. Fortunately, you will not need to worry too much about it until we start working with objects in a more serious way, in Chapter 5.

By the way, since class names and variable names are used in similar ways, it might be hard to tell which is which. Remember that all the built-in, predefined names in Java follow the rule that class names begin with an upper case letter while variable names begin with a lower case letter. While this is not a formal syntax rule, I strongly recommend that you follow it in your own programming. Subroutine names should also begin with lower case letters. There is no possibility of confusing a variable with a subroutine, since a subroutine name in a program is always followed by a left parenthesis.

As one final general note, you should be aware that subroutines in Java are often referred to as methods. Generally, the term “method” means a subroutine that is contained in a class or in an object. Since this is true of every subroutine in Java, every subroutine in Java is a method. The same is not true for other programming languages, and for the time being, I will prefer to use the more general term, “subroutine.” However, I should note that some people prefer to use the term “method” from the beginning.

34 *CHAPTER 2. NAMES AND THINGS*

2.3.3 Operations on Strings

String is a class, and a value of type String is an object. That object contains data, namely the sequence of characters that make up the string. It also contains subroutines. All of these subroutines are in fact functions. For example, every string object contains a function named length that computes the number of characters in that string. Suppose that advice is a variable that refers to a String. For example, advice might have been declared and assigned a value as follows:

String advice;

advice = "Seize the day!";

Then advice.length() is a function call that returns the number of characters in the string “Seize the day!”. In this case, the return value would be 14. In general, for any variable str of type String, the value of str.length() is an int equal to the number of characters in the string. Note that this function has no parameter; the particular string whose length is being computed is the value of str. The length subroutine is defined by the class String, and it can be used with any value of type String. It can even be used with String literals, which are, after all, just constant values of type String. For example, you could have a program count the characters in “Hello World” for you by saying

System.out.print("The number of characters in ");

System.out.print("the string \"Hello World\" is ");

System.out.println( "Hello World".length() );

The String class defines a lot of functions. Here are some that you might find useful. Assume that s1 and s2 are variables of type String:

• s1.equals(s2) is a function that returns a boolean value. It returns true if s1 consists of exactly the same sequence of characters as s2, and returns false otherwise. • s1.equalsIgnoreCase(s2) is another boolean-valued function that checks whether s1 is the same string as s2, but this function considers upper and lower case letters to be equivalent. Thus, if s1 is “cat”, then s1.equals("Cat") is false, while s1.equalsIgnoreCase("Cat") is true.

• s1.length(), as mentioned above, is an integer-valued function that gives the number of characters in s1.

• s1.charAt(N), where N is an integer, returns a value of type char. It returns the Nth character in the string. Positions are numbered starting with 0, so s1.charAt(0) is actually the first character, s1.charAt(1) is the second, and so on. The final position is s1.length() - 1. For example, the value of "cat".charAt(1) is ’a’. An error occurs if the value of the parameter is less than zero or is greater than or equal to s1.length().

• s1.substring(N,M), where N and M are integers, returns a value of type String. The returned value consists of the characters of s1 in positions N, N+1,. . . , M-1. Note that the character in position M is not included. The returned value is called a substring of s1. The subroutine s1.substring(N) returns the substring of s1 consisting of characters starting at position N up until the end of the string.

• s1.indexOf(s2) returns an integer. If s2 occurs as a substring of s1, then the returned value is the starting position of that substring. Otherwise, the returned value is -1. You can also use s1.indexOf(ch) to search for a char, ch, in s1. To find the first occurrence of x at or after position N, you can use s1.indexOf(x,N). To find the last occurance of x in s1, use s1.lastIndexOf(x).

*2.3. OBJECTS AND SUBROUTINES* 35

• s1.compareTo(s2) is an integer-valued function that compares the two strings. If the strings are equal, the value returned is zero. If s1 is less than s2, the value returned is a number less than zero, and if s1 is greater than s2, the value returned is some number greater than zero. (If both of the strings consist entirely of lower case letters, or if they consist entirely of upper case letters, then “less than” and “greater than” refer to alphabetical order. Otherwise, the ordering is more complicated.)

• s1.toUpperCase() is a String-valued function that returns a new string that is equal to s1, except that any lower case letters in s1 have been converted to upper case. For example, "Cat".toUpperCase() is the string "CAT". There is also a function s1.toLowerCase().

• s1.trim() is a String-valued function that returns a new string that is equal to s1 except that any non-printing characters such as spaces and tabs have been trimmed from the beginning and from the end of the string. Thus, if s1 has the value "fred ", then s1.trim() is the string "fred", with the spaces at the end removed.

For the functions s1.toUpperCase(), s1.toLowerCase(), and s1.trim(), note that the value of s1 is not changed. Instead a new string is created and returned as the value of the function. The returned value could be used, for example, in an assignment statement such as “smallLetters = s1.toLowerCase();”. To change the value of s1, you could use an assignment “s1 = s1.toLowerCase();”.

∗ ∗ ∗

Here is another extremely useful fact about strings: You can use the plus operator, +, to concatenate two strings. The concatenation of two strings is a new string consisting of all the characters of the first string followed by all the characters of the second string. For example, "Hello" + "World" evaluates to "HelloWorld". (Gotta watch those spaces, of course—if you want a space in the concatenated string, it has to be somewhere in the input data, as in "Hello " + "World".)

Let’s suppose that name is a variable of type String and that it already refers to the name of the person using the program. Then, the program could greet the user by executing the statement:

System.out.println("Hello, " + name + ". Pleased to meet you!");

Even more surprising is that you can actually concatenate values of any type onto a String using the + operator. The value is converted to a string, just as it would be if you printed it to the standard output, and then that string is concatenated with the other string. For example, the expression "Number" + 42 evaluates to the string "Number42". And the statements

System.out.print("After ");

System.out.print(years);

System.out.print(" years, the value is ");

System.out.print(principal);

can be replaced by the single statement:

System.out.print("After " + years +

" years, the value is " + principal);

Obviously, this is very convenient. It would have shortened some of the examples presented earlier in this chapter.

36 *CHAPTER 2. NAMES AND THINGS*

2.3.4 Introduction to Enums

Java comes with eight built-in primitive types and a large set of types that are defined by classes, such as String. But even this large collection of types is not sufficient to cover all the possible situations that a programmer might have to deal with. So, an essential part of Java, just like almost any other programming language, is the ability to create new types. For the most part, this is done by defining new classes; you will learn how to do that in Chapter 5. But we will look here at one particular case: the ability to define enums (short for enumerated types).

Technically, an enum is considered to be a special kind of class, but that is not important for now. In this section, we will look at enums in a simplified form. In practice, most uses of enums will only need the simplified form that is presented here.

An enum is a type that has a fixed list of possible values, which is specified when the enum is created. In some ways, an enum is similar to the boolean data type, which has true and false as its only possible values. However, boolean is a primitive type, while an enum is not. The definition of an enum type has the (simplified) form:

enum h*enum-type-name* i { h*list-of-enum-values* i }

This definition cannot be inside a subroutine. You can place it outside the main() routine of the program. The henum-type-namei can be any simple identifier. This identifier becomes the name of the enum type, in the same way that “boolean” is the name of the boolean type and “String” is the name of the String type. Each value in the hlist-of-enum-valuesi must be a simple identifier, and the identifiers in the list are separated by commas. For example, here is the definition of an enum type named Season whose values are the names of the four seasons of the year:

enum Season { SPRING, SUMMER, FALL, WINTER }

By convention, enum values are given names that are made up of upper case letters, but that is a style guideline and not a syntax rule. An enum value is a constant; that is, it represents a fixed value that cannot be changed. The possible values of an enum type are usually referred to as enum constants.

Note that the enum constants of type Season are considered to be “contained in” Season, which means—following the convention that compound identifiers are used for things that are contained in other things—the names that you actually use in your program to refer to them are Season.SPRING, Season.SUMMER, Season.FALL, and Season.WINTER.

Once an enum type has been created, it can be used to declare variables in exactly the same ways that other types are used. For example, you can declare a variable named vacation of type Season with the statement:

Season vacation;

After declaring the variable, you can assign a value to it using an assignment statement. The value on the right-hand side of the assignment can be one of the enum constants of type Season. Remember to use the full name of the constant, including “Season”! For example:

vacation = Season.SUMMER;

You can print out an enum value with an output statement such as System.out.print(vacation). The output value will be the name of the enum constant (without the “Season.”). In this case, the output would be “SUMMER”.

*2.4. TEXT INPUT AND OUTPUT* 37

Because an enum is technically a class, the enum values are technically objects. As ob jects, they can contain subroutines. One of the subroutines in every enum value is named ordinal(). When used with an enum value, it returns the ordinal number of the value in the list of values of the enum. The ordinal number simply tells the position of the value in the list. That is, Season.SPRING.ordinal() is the int value 0, Season.SUMMER.ordinal() is 1, Season.FALL.ordinal() is 2, and Season.WINTER.ordinal() is 3. (You will see over and over again that computer scientists like to start counting at zero!) You can, of course, use the ordinal() method with a variable of type Season, such as vacation.ordinal().

Using enums can make a program more readable, since you can use meaningful names for the values. And it can prevent certain types of errors, since a compiler can check that the values assigned to an enum variable are in fact legal values for that variable. However, we will in fact use them only occasionally in this book. For now, you should just appreciate them as the first example of an important concept: creating new types. Here is a little example that shows enums being used in a complete program:

public class EnumDemo {

// Define two enum types -- remember that the definitions

// go OUTSIDE The main() routine!

enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY } enum Month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC } public static void main(String[] args) {

Day tgif; // Declare a variable of type Day.

Month libra; // Declare a variable of type Month.

tgif = Day.FRIDAY; // Assign a value of type Day to tgif.

libra = Month.OCT; // Assign a value of type Month to libra.

System.out.print("My sign is libra, since I was born in ");

System.out.println(libra); // Output value will be: OCT

System.out.print("That’s the ");

System.out.print( libra.ordinal() );

System.out.println("-th month of the year.");

System.out.println(" (Counting from 0, of course!)");

System.out.print("Isn’t it nice to get to ");

System.out.println(tgif); // Output value will be: FRIDAY

System.out.println( tgif + " is the " + tgif.ordinal()

+ "-th day of the week.");

}

}

2.4 Text Input and Output

We have seen that it is very easy to display text to the user with the functions System.out.print and System.out.println. But there is more to say on the topic of out putting text. Furthermore, most programs use data that is input to the program at run time, so you need to know how to do input as well as output. This section explains how to get data

38 *CHAPTER 2. NAMES AND THINGS*

from the user, and it covers output in more detail than we have seen so far. It also has a section on using files for input and output.

2.4.1 Basic Output and Formatted Output

The most basic output function is System.out.print(x), where x can be a value or expression of any type. If the parameter, x, is not already a string, it is converted to a value of type String, and the string is then output to the destination called standard output. (Generally, this means that the string is displayed to the user; however, in GUI programs, it outputs to a place where a typical user is unlikely to see it. Furthermore, standard output can be “redirected” to write to a different output destination. Nevertheless, for the type of program that we are working with now, the purpose of System.out is to display text to the user.)

System.out.println(x) outputs the same text as System.out.print, but it follows that text by a line feed, which means that any subsequent output will be on the next line. It is possible to use this function with no parameter, System.out.println(), which outputs nothing but a line feed. Note that System.out.println(x) is equivalent to

System.out.print(x);

System.out.println();

You might have noticed that System.out.print outputs real numbers with as many digits after the decimal point as necessary, so that for example π is output as 3.141592653589793, and numbers that are supposed to represent money might be output as 1050.0 or 43.575. You might prefer to have these numbers output as, for example, 3.14159, 1050.00, and 43.58. Java has a “formatted output” capability that makes it easy to control how real numbers and other values are printed. A lot of formatting options are available. I will cover just a few of the simplest and most commonly used possibilities here.

The function System.out.printf can be used to produce formatted output. (The name “printf,” which stands for “print formatted,” is copied from the C and C++ programming languages, where this type of output originated.) System.out.printf takes one or more pa rameters. The first parameter is a String that specifies the format of the output. This parameter is called the format string. The remaining parameters specify the values that are to be out put. Here is a statement that will print a number in the proper format for a dollar amount, where amount is a variable of type double:

System.out.printf( "%1.2f", amount );

The output format of a value is specified by a format specifier. In this example, the format specifier is %1.2f. The format string (in the simple cases that I cover here) contains one format specifier for each of the values that is to be output. Some typical format specifiers are %d, %12d, %10s, %1.2f, %15.8e and %1.8g. Every format specifier begins with a percent sign (%) and ends with a letter, possibly with some extra formatting information in between. The letter specifies the type of output that is to be produced. For example, in %d and %12d, the “d” specifies that an integer is to be written. The “12” in %12d specifies the minimum number of spaces that should be used for the output. If the integer that is being output takes up fewer than 12 spaces, extra blank spaces are added in front of the integer to bring the total up to 12. We say that the output is “right-justified in a field of length 12.” A very large value is not forced into 12 spaces; if the value has more than 12 digits, all the digits will be printed, with no extra spaces. The specifier %d means the same as %1d—that is, an integer will be printed using just as many spaces as necessary. (The “d,” by the way, stands for “decimal”—that is, base-10—numbers. You can replace the “d” with an “x” to output an integer value in hexadecimal form.)

*2.4. TEXT INPUT AND OUTPUT* 39

The letter “s” at the end of a format specifier can be used with any type of value. It means that the value should be output in its default format, just as it would be in unformatted output. A number, such as the “20” in %20s, can be added to specify the (minimum) number of characters. The “s” stands for “string,” and it can be used for values of type String. It can also be used for values of other types; in that case the value is converted into a String value in the usual way.

The format specifiers for values of type double are more complicated. An “f”, as in %1.2f, is used to output a number in “floating-point” form, that is with digits after a decimal point. In %1.2f, the “2” specifies the number of digits to use after the decimal point. The “1” specifies the (minimum) number of characters to output; a “1” in this position effectively means that just as many characters as are necessary should be used. Similarly, %12.3f would specify a floating-point format with 3 digits after the decimal point, right-justified in a field of length 12.

Very large and very small numbers should be written in exponential format, such as 6.00221415e23, representing “6.00221415 times 10 raised to the power 23.” A format speci fier such as %15.8e specifies an output in exponential form, with the “8” telling how many digits to use after the decimal point. If you use “g” instead of “e”, the output will be in ex ponential form for very small values and very large values and in floating-point form for other values. In %1.8g, the 8 gives the total number of digits in the answer, including both the digits before the decimal point and the digits after the decimal point.

For numeric output, the format specifier can include a comma (“,”), which will cause the digits of the number to be separated into groups, to make it easier to read big numbers. In the United States, groups of three digits are separated by commas. For example, if x is one billion, then System.out.printf("%,d",x) will output 1,000,000,000. In other countries, the separator character and the number of digits per group might be different. The comma should come at the beginning of the format specifier, before the field width; for example: %,12.3f. If you want the output to be left-justified instead of right justified, add a minus sign to the beginning of the format specifier: for example, %-20s.

In addition to format specifiers, the format string in a printf statement can include other characters. These extra characters are just copied to the output. This can be a convenient way to insert values into the middle of an output string. For example, if x and y are variables of type int, you could say

System.out.printf("The product of %d and %d is %d", x, y, x\*y);

When this statement is executed, the value of x is substituted for the first %d in the string, the value of y for the second %d, and the value of the expression x\*y for the third, so the output would be something like “The product of 17 and 42 is 714” (quotation marks not included in output!).

To output a percent sign, use the format specifier %% in the format string. You can use %n to output a line feed. You can also use a backslash, \, as usual in strings to output special characters such as tabs and double quote characters.

2.4.2 A First Text Input Example

For some unfathomable reason, Java has never made it very easy to read data typed in by the user of a program. You’ve already seen that output can be displayed to the user using the sub routine System.out.print. This subroutine is part of a pre-defined object called System.out. The purpose of this object is precisely to display output to the user. There is a correspond ing object called System.in that exists to read data input by the user, but it provides only

40 *CHAPTER 2. NAMES AND THINGS*

very primitive input facilities, and it requires some advanced Java programming skills to use it effectively.

Java 5.0 finally made input a little easier with a new Scanner class. However, it requires some knowledge of object-oriented programming to use this class, so it’s not ideal for use here at the beginning of this course. Java 6 introduced the Console class for communicating with the user, but Console has its own problems. (It is not always available, and it can only read strings, not numbers.) Furthermore, in my opinion, Scanner and Console still don’t get things quite right. Nevertheless, I will introduce Scanner briefly at the end of this section, in case you want to start using it now. However, we start with my own version of text input.

Fortunately, it is possible to extend Java by creating new classes that provide subroutines that are not available in the standard part of the language. As soon as a new class is available, the subroutines that it contains can be used in exactly the same way as built-in routines. Along these lines, I’ve written a class named TextIO that defines subroutines for reading values typed by the user. The subroutines in this class make it possible to get input from the standard input object, System.in, without knowing about the advanced aspects of Java that are needed to use Scanner or to use System.in directly. TextIO also has a few other capabilities that I will discuss later in this section.

To use the TextIO class, you must make sure that the class is available to your program. What this means depends on the Java programming environment that you are using. In general, you just have to add the source code file, TextIO.java, to the same directory that contains your main program. See Section 2.6 for information about how to use TextIO.

The input routines in the TextIO class are static member functions. (Static member func tions were introduced in the previous section.) Let’s suppose that you want your program to read an integer typed in by the user. The TextIO class contains a static member function named getlnInt that you can use for this purpose. Since this function is contained in the TextIO class, you have to refer to it in your program as TextIO.getlnInt. The function has no parameters, so a complete call to the function takes the form “TextIO.getlnInt()”. This function call represents the int value typed by the user, and you have to do something with the returned value, such as assign it to a variable. For example, if userInput is a variable of type int (created with a declaration statement “int userInput;”), then you could use the assignment statement

userInput = TextIO.getlnInt();

When the computer executes this statement, it will wait for the user to type in an integer value. The user must type a number and press return before the program can continue. The value that the user typed will then be returned by the function, and it will be stored in the variable, userInput. Here is a complete program that uses TextIO.getlnInt to read a number typed by the user and then prints out the square of that number:

/\*\*

\* A program that reads an integer that is typed in by the

\* user and computes and prints the square of that integer.

\*/

public class PrintSquare {

public static void main(String[] args) {

int userInput; // The number input by the user.

int square; // The userInput, multiplied by itself.

*2.4. TEXT INPUT AND OUTPUT* 41

System.out.print("Please type a number: ");

userInput = TextIO.getlnInt();

square = userInput \* userInput;

System.out.println();

System.out.println("The number that you entered was " + userInput); System.out.println("The square of that number is " + square);

System.out.println();

} // end of main()

} //end of class PrintSquare

When you run this program, it will display the message “Please type a number:” and will pause until you type a response, including a carriage return after the number. Note that it is good style to output a question or some other prompt to the user before reading input. Otherwise, the user will have no way of knowing exactly what the computer is waiting for, or even that it is waiting for the user to do something.

2.4.3 Basic TextIO Input Functions

TextIO includes a variety of functions for inputting values of various types. Here are the functions that you are most likely to use:

j = TextIO.getlnInt(); // Reads a value of type int.

y = TextIO.getlnDouble(); // Reads a value of type double.

a = TextIO.getlnBoolean(); // Reads a value of type boolean.

c = TextIO.getlnChar(); // Reads a value of type char.

w = TextIO.getlnWord(); // Reads one "word" as a value of type String. s = TextIO.getln(); // Reads an entire input line as a String.

For these statements to be legal, the variables on the left side of each assignment statement must already be declared and must be of the same type as that returned by the function on the right side. Note carefully that these functions do not have parameters. The values that they return come from outside the program, typed in by the user as the program is running. To “capture” that data so that you can use it in your program, you have to assign the return value of the function to a variable. You will then be able to refer to the user’s input value by using the name of the variable.

When you call one of these functions, you are guaranteed that it will return a legal value of the correct type. If the user types in an illegal value as input—for example, if you ask for an int and the user types in a non-numeric character or a number that is outside the legal range of values that can be stored in a variable of type int—then the computer will ask the user to re-enter the value, and your program never sees the first, illegal value that the user entered. For TextIO.getlnBoolean(), the user is allowed to type in any of the following: true, false, t, f, yes, no, y, n, 1, or 0. Furthermore, they can use either upper or lower case letters. In any case, the user’s input is interpreted as a true/false value. It’s convenient to use TextIO.getlnBoolean() to read the user’s response to a Yes/No question.

You’ll notice that there are two input functions that return Strings. The first, getlnWord(), returns a string consisting of non-blank characters only. When it is called, it skips over any spaces and carriage returns typed in by the user. Then it reads non-blank characters until it gets to the next space or carriage return. It returns a String consisting of all the non blank characters that it has read. The second input function, getln(), simply returns a string consisting of all the characters typed in by the user, including spaces, up to the next carriage

42 *CHAPTER 2. NAMES AND THINGS*

return. It gets an entire line of input text. The carriage return itself is not returned as part of the input string, but it is read and discarded by the computer. Note that the String returned by TextIO.getln() might be the empty string, "", which contains no characters at all. You will get this return value if the user simply presses return, without typing anything else first.

TextIO.getln() does not skip blanks or end-of-lines before reading a value. But the input functions getlnInt(), getlnDouble(), getlnBoolean(), and getlnChar() behave like getlnWord() in that they will skip past any blanks and carriage returns in the input before reading a value. When one of these functions skips over an end-of-line, it outputs a ’?’ to let the user know that more input is expected.

Furthermore, if the user types extra characters on the line after the input value, all the extra characters will be discarded, along with the carriage return at the end of the line. If the program executes another input function, the user will have to type in another line of input, even if they had typed more than one value on the previous line. It might not sound like a good idea to discard any of the user’s input, but it turns out to be the safest thing to do in most programs.

∗ ∗ ∗

Using TextIO for input and output, we can now improve the program from Section 2.2 for computing the value of an investment. We can have the user type in the initial value of the investment and the interest rate. The result is a much more useful program—for one thing, it makes sense to run it more than once! Note that this program uses formatted output to print out monetary values in their correct format.

/\*\*

\* This class implements a simple program that will compute

\* the amount of interest that is earned on an investment over

\* a period of one year. The initial amount of the investment

\* and the interest rate are input by the user. The value of

\* the investment at the end of the year is output. The

\* rate must be input as a decimal, not a percentage (for

\* example, 0.05 rather than 5).

\*/

public class Interest2 {

public static void main(String[] args) {

double principal; // The value of the investment.

double rate; // The annual interest rate.

double interest; // The interest earned during the year.

System.out.print("Enter the initial investment: ");

principal = TextIO.getlnDouble();

System.out.print("Enter the annual interest rate (as a decimal): "); rate = TextIO.getlnDouble();

interest = principal \* rate; // Compute this year’s interest. principal = principal + interest; // Add it to principal.

System.out.printf("The amount of interest is $%1.2f%n", interest); System.out.printf("The value after one year is $%1.2f%n", principal);

} // end of main()

} // end of class Interest2

*2.4. TEXT INPUT AND OUTPUT* 43

(You might be wondering why there is only one output routine, System.out.println, which can output data values of any type, while there is a separate input routine for each data type. For the output function, the computer can tell what type of value is being output by looking at the parameter. However, the input routines don’t have parameters, so the different input routines can only be distinguished by having different names.)

2.4.4 Introduction to File I/O

System.out sends its output to the output destination known as “standard output.” But stan dard output is just one possible output destination. For example, data can be written to a file that is stored on the user’s hard drive. The advantage to this, of course, is that the data is saved in the file even after the program ends, and the user can print the file, email it to someone else, edit it with another program, and so on. Similarly, System.in has only one possible source for input data.

TextIO has the ability to write data to files and to read data from files. TextIO includes output functions TextIO.put, TextIO.putln, and TextIO.putf. Ordinarily, these functions work exactly like System.out.print, System.out.println, and System.out.printf and are interchangeable with them. However, they can also be used to output text to files and to other destinations.

When you write output using TextIO.put, TextIO.putln, or TextIO.putf, the output is sent to the current output destination. By default, the current output destination is standard output. However, TextIO has subroutines that can be used to change the current output destination. To write to a file named “result.txt”, for example, you would use the statement:

TextIO.writeFile("result.txt");

After this statement is executed, any output from TextIO output statements will be sent to the file named “result.txt” instead of to standard output. The file will be created if it does not already exist. Note that if a file with the same name already exists, its previous contents will be erased without any warning!

When you call TextIO.writeFile, TextIO remembers the file and automatically sends any output from TextIO.put or other output functions to that file. If you want to go back to writing to standard output, you can call

TextIO.writeStandardOutput();

Here is a simple program that asks the user some questions and outputs the user’s responses to a file named “profile.txt.” As an example, it uses TextIO for output to standard output as well as to the file, but System.out could also have been used for the output to standard output.

public class CreateProfile {

public static void main(String[] args) {

String name; // The user’s name.

String email; // The user’s email address.

double salary; // the user’s yearly salary.

String favColor; // The user’s favorite color.

TextIO.putln("Good Afternoon! This program will create");

TextIO.putln("your profile file, if you will just answer");

TextIO.putln("a few simple questions.");

44 *CHAPTER 2. NAMES AND THINGS*

TextIO.putln();

/\* Gather responses from the user. \*/

TextIO.put("What is your name? ");

name = TextIO.getln();

TextIO.put("What is your email address? ");

email = TextIO.getln();

TextIO.put("What is your yearly income? ");

salary = TextIO.getlnDouble();

TextIO.put("What is your favorite color? ");

favColor = TextIO.getln();

/\* Write the user’s information to the file named profile.txt. \*/

TextIO.writeFile("profile.txt"); // subsequent output goes to file TextIO.putln("Name: " + name);

TextIO.putln("Email: " + email);

TextIO.putln("Favorite Color: " + favColor);

TextIO.putf( "Yearly Income: %,1.2f%n", salary);

/\* Print a final message to standard output. \*/

TextIO.writeStandardOutput();

TextIO.putln("Thank you. Your profile has been written to profile.txt."); }

}

In many cases, you want to let the user select the file that will be used for output. You could ask the user to type in the file name, but that is error-prone, and users are more familiar with selecting a file from a file dialog box. The statement

TextIO.writeUserSelectedFile();

will open a typical graphical-user-interface file selection dialog where the user can specify the output file. This also has the advantage of alerting the user if they are about to replace an existing file. It is possible for the user to cancel the dialog box without selecting a file. TextIO.writeUserSelectedFile is a function that returns a boolean value. The return value is true if the user selected a file, and is false if the user canceled the dialog box. Your program can check the return value if it needs to know whether it is actually going to write to a file or not.

∗ ∗ ∗

TextIO can also read from files, as an alternative to reading from standard input. You can specify an input source for TextIO’s various “get” functions. The default input source is standard input. You can use the statement TextIO.readFile("data.txt") to read from a file named “data.txt” instead, or you can let the user select the input file with a GUI-style dialog box by saying TextIO.readUserSelectedFile(). After you have done this, any input will come from the file instead of being typed by the user. You can go back to reading the user’s input with TextIO.readStandardInput().

When your program is reading from standard input, the user gets a chance to correct any errors in the input. This is not possible when the program is reading from a file. If illegal data is found when a program tries to read from a file, an error occurs that will crash the program.

*2.4. TEXT INPUT AND OUTPUT* 45

(Later, we will see that it is possible to “catch” such errors and recover from them.) Errors can also occur, though more rarely, when writing to files.

A complete understanding of input/output in Java requires a knowledge of object oriented programming. We will return to the topic later, in Chapter 11. The file I/O capabilities in Tex tIO are rather primitive by comparison. Nevertheless, they are sufficient for many applications, and they will allow you to get some experience with files sooner rather than later.

2.4.5 Other TextIO Features

The TextIO input functions that we have seen so far can only read one value from a line of input. Sometimes, however, you do want to read more than one value from the same line of input. For example, you might want the user to be able to type something like “42 17” to input the two numbers 42 and 17 on the same line. TextIO provides the following alternative input functions to allow you to do this:

j = TextIO.getInt(); // Reads a value of type int.

y = TextIO.getDouble(); // Reads a value of type double.

a = TextIO.getBoolean(); // Reads a value of type boolean.

c = TextIO.getChar(); // Reads a value of type char.

w = TextIO.getWord(); // Reads one "word" as a value of type String.

The names of these functions start with “get” instead of “getln”. “Getln” is short for “get line” and should remind you that the functions whose names begin with “getln” will consume an entire line of data. A function without the “ln” will read an input value in the same way, but will then save the rest of the input line in a chunk of internal memory called the input buffer. The next time the computer wants to read an input value, it will look in the input buffer before prompting the user for input. This allows the computer to read several values from one line of the user’s input. Strictly speaking, the computer actually reads only from the input buffer. The first time the program tries to read input from the user, the computer will wait while the user types in an entire line of input. TextIO stores that line in the input buffer until the data on the line has been read or discarded (by one of the “getln” functions). The user only gets to type when the buffer is empty.

Note, by the way, that although the TextIO input functions will skip past blank spaces and carriage returns while looking for input, they will not skip past other characters. For example, if you try to read two ints and the user types “42,17”, the computer will read the first number correctly, but when it tries to read the second number, it will see the comma. It will regard this as an error and will force the user to retype the number. If you want to input several numbers from one line, you should make sure that the user knows to separate them with spaces, not commas. Alternatively, if you want to require a comma between the numbers, use getChar() to read the comma before reading the second number.

There is another character input function, TextIO.getAnyChar(), which does not skip past blanks or carriage returns. It simply reads and returns the next character typed by the user, even if it’s a blank or carriage return. If the user typed a carriage return, then the char returned by getAnyChar() is the special linefeed character ’\n’. There is also a function, TextIO.peek(), that lets you look ahead at the next character in the input without actually reading it. After you “peek” at the next character, it will still be there when you read the next item from input. This allows you to look ahead and see what’s coming up in the input, so that you can take different actions depending on what’s there.

The TextIO class provides a number of other functions. To learn more about them, you can look at the comments in the source code file, TextIO.java.

46 *CHAPTER 2. NAMES AND THINGS*

Clearly, the semantics of input is much more complicated than the semantics of output! Fortunately, for the majority of applications, it’s pretty straightforward in practice. You only need to follow the details if you want to do something fancy. In particular, I strongly advise you to use the “getln” versions of the input routines, rather than the “get” versions, unless you really want to read several items from the same line of input, precisely because the semantics of the “getln” versions is much simpler.

2.4.6 Using Scanner for Input

TextIO makes it easy to get input from the user. However, since it is not a standard class, you have to remember to make TextIO.java available to any program that uses it. Another option for input is the Scanner class. One advantage of using Scanner is that it’s a standard part of Java and so is always there when you want it.

It’s not that hard to use a Scanner for user input, and it has some nice features, but using it requires some syntax that will not be introduced until Chapter 4 and Chapter 5. I’ll tell you how to do it here, without explaining why it works. You won’t understand all the syntax at this point. (Scanners will be covered in more detail in Subsection 11.1.5.)

First, you should add the following line to your program at the beginning of the source code file, before the “public class. . . ”:

import java.util.Scanner;

Then include the following statement at the beginning of your main() routine: Scanner stdin = new Scanner( System.in );

This creates a variable named stdin of type Scanner. (You can use a different name for the variable if you want; “stdin” stands for “standard input.”) You can then use stdin in your program to access a variety of subroutines for reading user input. For example, the function stdin.nextInt() reads one value of type int from the user and returns it. It is almost the same as TextIO.getInt() except for two things: If the value entered by the user is not a legal int, then stdin.nextInt() will crash rather than prompt the user to re-enter the value. And the integer entered by the user must be followed by a blank space or by an end-of-line, whereas TextIO.getInt() will stop reading at any character that is not a digit.

There are corresponding methods for reading other types of data, including stdin.nextDouble(), stdin.nextLong(), and stdin.nextBoolean(). (stdin.nextBoolean() will only accept “true” or “false” as input.) These subroutines can read more than one value from a line, so they are more similar to the “get” versions of TextIO subroutines rather than the “getln” versions. The method stdin.nextLine() is equivalent to TextIO.getln(), and stdin.next(), like TextIO.getWord(), returns a string of non-blank characters.

As a simple example, here is a version of the sample program Interest2.java that uses Scanner instead of TextIO for user input:

*import java.util.Scanner;* // Make the Scanner class available.

public class Interest2WithScanner {

public static void main(String[] args) {

*Scanner stdin = new Scanner( System.in );* // Create the Scanner.

double principal; // The value of the investment.

double rate; // The annual interest rate.

double interest; // The interest earned during the year.

*2.5. DETAILS OF EXPRESSIONS* 47

System.out.print("Enter the initial investment: ");

*principal = stdin.nextDouble();*

System.out.print("Enter the annual interest rate (as a decimal): "); *rate = stdin.nextDouble();*

interest = principal \* rate; // Compute this year’s interest. principal = principal + interest; // Add it to principal.

System.out.printf("The amount of interest is $%1.2f%n", interest); System.out.printf("The value after one year is $%1.2f%n", principal);

} // end of main()

} // end of class Interest2With Scanner

Note the inclusion of the two lines given above to import Scanner and create stdin. Also note the substitution of stdin.nextDouble() for TextIO.getlnDouble(). (In fact, stdin.nextDouble() is really equivalent to TextIO.getDouble() rather than to the “getln” version, but this will not affect the behavior of the program as long as the user types just one number on each line of input.)

I will continue to use TextIO for input for the time being, but I will give a few more examples of using Scanner in the on-line solutions to the end-of-chapter exercises. There will be more detailed coverage of Scanner later in the book.

2.5 Details of Expressions

This section takes a closer look at expressions. Recall that an expression is a piece of program code that represents or computes a value. An expression can be a literal, a variable, a function call, or several of these things combined with operators such as + and >. The value of an expression can be assigned to a variable, used as a parameter in a subroutine call, or combined with other values into a more complicated expression. (The value can even, in some cases, be ignored, if that’s what you want to do; this is more common than you might think.) Expressions are an essential part of programming. So far, this book has dealt only informally with expressions. This section tells you the more-or-less complete story (leaving out some of the less commonly used operators).

The basic building blocks of expressions are literals (such as 674, 3.14, true, and ’X’), variables, and function calls. Recall that a function is a subroutine that returns a value. You’ve already seen some examples of functions, such as the input routines from the TextIO class and the mathematical functions from the Math class.

The Math class also contains a couple of mathematical constants that are useful in mathematical expressions: Math.PI represents π (the ratio of the circumference of a cir cle to its diameter), and Math.E represents e (the base of the natural logarithms). These “constants” are actually member variables in Math of type double. They are only ap proximations for the mathematical constants, which would require an infinite number of digits to specify exactly. The standard class Integer contains a couple of constants re lated to the int data type: Integer.MAX VALUE is the largest possible int, 2147483647, and Integer.MIN VALUE is the smallest int, -2147483648. Similarly, the class Double contains some constants related to type double. Double.MAX VALUE is the largest value of type double, and Double.MIN VALUE is the smallest positive value. It also has constants to represent infinite values, Double.POSITIVE INFINITY and Double.NEGATIVE INFINITY, and the special value

48 *CHAPTER 2. NAMES AND THINGS*

Double.NaN to represent an undefined value. For example, the value of Math.sqrt(-1) is Double.NaN.

Literals, variables, and function calls are simple expressions. More complex expressions can be built up by using operators to combine simpler expressions. Operators include + for adding two numbers, > for comparing two values, and so on. When several operators appear in an expression, there is a question of precedence, which determines how the operators are grouped for evaluation. For example, in the expression “A + B \* C”, B\*C is computed first and then the result is added to A. We say that multiplication (\*) has higher precedence than addition (+). If the default precedence is not what you want, you can use parentheses to explicitly specify the grouping you want. For example, you could use “(A + B) \* C” if you want to add A to B first and then multiply the result by C.

The rest of this section gives details of operators in Java. The number of operators in Java is quite large. I will not cover them all here, but most of the important ones are here.

2.5.1 Arithmetic Operators

Arithmetic operators include addition, subtraction, multiplication, and division. They are indicated by +, -, \*, and /. These operations can be used on values of any numeric type: byte, short, int, long, float, or double. (They can also be used with values of type char, which are treated as integers in this context; a char is converted into its Unicode code number when it is used with an arithmetic operator.) When the computer actually calculates one of these operations, the two values that it combines must be of the same type. If your program tells the computer to combine two values of different types, the computer will convert one of the values from one type to another. For example, to compute 37.4 + 10, the computer will convert the integer 10 to a real number 10.0 and will then compute 37.4 + 10.0. This is called a type conversion. Ordinarily, you don’t have to worry about type conversion in expressions, because the computer does it automatically.

When two numerical values are combined (after doing type conversion on one of them, if necessary), the answer will be of the same type. If you multiply two ints, you get an int; if you multiply two doubles, you get a double. This is what you would expect, but you have to be very careful when you use the division operator /. When you divide two integers, the answer will always be an integer; if the quotient has a fractional part, it is discarded. For example, the value of 7/2 is 3, not 3.5. If N is an integer variable, then N/100 is an integer, and 1/N is equal to zero for any N greater than one! This fact is a common source of programming errors. You can force the computer to compute a real number as the answer by making one of the operands real: For example, when the computer evaluates 1.0/N, it first converts N to a real number in order to match the type of 1.0, so you get a real number as the answer.

Java also has an operator for computing the remainder when one number is divided by another. This operator is indicated by %. If A and B are integers, then A % B represents the remainder when A is divided by B. (However, for negative operands, % is not quite the same as the usual mathematical “modulus” operator, since if one of A or B is negative, then the value of A % B will be negative.) For example, 7 % 2 is 1, while 34577 % 100 is 77, and 50 % 8 is 2. A common use of % is to test whether a given integer is even or odd: N is even if N % 2 is zero, and it is odd if N % 2 is 1. More generally, you can check whether an integer N is evenly divisible by an integer M by checking whether N % M is zero.

The % operator also works with real numbers. In general, A % B is what is left over after you remove as many copies of B as possible from A. For example, 7.52 % 0.5 is 0.02.

*2.5. DETAILS OF EXPRESSIONS* 49

Finally, you might need the unary minus operator, which takes the negative of a number. For example, -X has the same value as (-1)\*X. For completeness, Java also has a unary plus operator, as in +X, even though it doesn’t really do anything.

By the way, recall that the + operator can also be used to concatenate a value of any type onto a String. When you use + to combine a string with a value of some other type, it is another example of type conversion, since any type can be automatically converted into type String.

2.5.2 Increment and Decrement

You’ll find that adding 1 to a variable is an extremely common operation in programming. Subtracting 1 from a variable is also pretty common. You might perform the operation of adding 1 to a variable with assignment statements such as:

counter = counter + 1;

goalsScored = goalsScored + 1;

The effect of the assignment statement x = x + 1 is to take the old value of the variable x, compute the result of adding 1 to that value, and store the answer as the new value of x. The same operation can be accomplished by writing x++ (or, if you prefer, ++x). This actually changes the value of x, so that it has the same effect as writing “x = x + 1”. The two statements above could be written

counter++;

goalsScored++;

Similarly, you could write x-- (or --x) to subtract 1 from x. That is, x-- performs the same computation as x = x - 1. Adding 1 to a variable is called incrementing that variable, and subtracting 1 is called decrementing. The operators ++ and -- are called the increment operator and the decrement operator, respectively. These operators can be used on variables belonging to any of the numerical types and also on variables of type char. (’A’++ is ’B’.)

Usually, the operators ++ or -- are used in statements like “x++;” or “x--;”. These state ments are commands to change the value of x. However, it is also legal to use x++, ++x, x--, or --x as expressions, or as parts of larger expressions. That is, you can write things like:

y = x++;

y = ++x;

TextIO.putln(--x);

z = (++x) \* (y--);

The statement “y = x++;” has the effects of adding 1 to the value of x and, in addition, assigning some value to y. The value assigned to y is the value of the expression x++, which is defined to be the old value of x, before the 1 is added. Thus, if the value of x is 6, the statement “y = x++;” will change the value of x to 7, but it will change the value of y to 6 since the value assigned to y is the old value of x. On the other hand, the value of ++x is defined to be the new value of x, after the 1 is added. So if x is 6, then the statement “y = ++x;” changes the values of both x and y to 7. The decrement operator, --, works in a similar way.

Note in particular that the statement x = x++; does not change the value of x! This is because the value that is being assigned to x is the old value of x, the one that it had before the statement was executed. The net result is that x is incremented but then immediately changed back to its previous value! You also need to remember that x++ is not the same as x + 1. The expression x++ changes the value of x; the expression x + 1 does not.

50 *CHAPTER 2. NAMES AND THINGS*

This can be confusing, and I have seen many bugs in student programs resulting from the confusion. My advice is: Don’t be confused. Use ++ and -- only as stand-alone statements, not as expressions. I will follow this advice in almost all examples in these notes.

2.5.3 Relational Operators

Java has boolean variables and boolean-valued expressions that can be used to express con ditions that can be either true or false. One way to form a boolean-valued expression is to compare two values using a relational operator. Relational operators are used to test whether two values are equal, whether one value is greater than another, and so forth. The relational operators in Java are: ==, !=, <, >, <=, and >=. The meanings of these operators are:

A == B Is A "equal to" B?

A != B Is A "not equal to" B?

A < B Is A "less than" B?

A > B Is A "greater than" B?

A <= B Is A "less than or equal to" B?

A >= B Is A "greater than or equal to" B?

These operators can be used to compare values of any of the numeric types. They can also be used to compare values of type char. For characters, < and > are defined according the numeric Unicode values of the characters. (This might not always be what you want. It is not the same as alphabetical order because all the upper case letters come before all the lower case letters.)

When using boolean expressions, you should remember that as far as the computer is con cerned, there is nothing special about boolean values. In the next chapter, you will see how to use them in loop and branch statements. But you can also assign boolean-valued expressions to boolean variables, just as you can assign numeric values to numeric variables. And functions can return boolean values.

By the way, the operators == and != can be used to compare boolean values too. This is occasionally useful. For example, can you figure out what this does:

boolean sameSign;

sameSign = ((x > 0) == (y > 0));

One thing that you cannot do with the relational operators <, >, <=, and >= is to use them to compare values of type String. You can legally use == and != to compare Strings, but because of peculiarities in the way objects behave, they might not give the results you want. (The == operator checks whether two objects are stored in the same memory location, rather than whether they contain the same value. Occasionally, for some objects, you do want to make such a check—but rarely for strings. I’ll get back to this in a later chapter.) Instead, you should use the subroutines equals(), equalsIgnoreCase(), and compareTo(), which were described in Subsection 2.3.3, to compare two Strings.

Another place where == and != don’t work as you would expect is with Double.NaN, the constant that represents an undefined value of type double. The values of x == Double.NaN and x != Double.NaN are both defined to be false in all cases, whether or not x is Double.NaN! To test whether a real value x is the undefined value Double.NaN, use the boolean-valued function Double.isNaN(x).

2.5.4 Boolean Operators

In English, complicated conditions can be formed using the words “and”, “or”, and “not.” For example, “If there is a test and you did not study for it. . . ”. “And”, “or”, and “not” are

*2.5. DETAILS OF EXPRESSIONS* 51

boolean operators, and they exist in Java as well as in English.

In Java, the boolean operator “and” is represented by &&. The && operator is used to combine two boolean values. The result is also a boolean value. The result is true if both of the combined values are true, and the result is false if either of the combined values is false. For example, “(x == 0) && (y == 0)” is true if and only if both x is equal to 0 and y is equal to 0.

The boolean operator “or” is represented by ||. (That’s supposed to be two of the vertical line characters, |.) The expression “A || B” is true if either A is true or B is true, or if both are true. “A || B” is false only if both A and B are false.

The operators && and || are said to be short-circuited versions of the boolean operators. This means that the second operand of && or || is not necessarily evaluated. Consider the test

(x != 0) && (y/x > 1)

Suppose that the value of x is in fact zero. In that case, the division y/x is undefined math ematically. However, the computer will never perform the division, since when the computer evaluates (x != 0), it finds that the result is false, and so it knows that ((x != 0) && any thing) has to be false. Therefore, it doesn’t bother to evaluate the second operand. The evaluation has been short-circuited and the division by zero is avoided. (This may seem like a technicality, and it is. But at times, it will make your programming life a little easier.)

The boolean operator “not” is a unary operator. In Java, it is indicated by ! and is written in front of its single operand. For example, if test is a boolean variable, then

test = ! test;

will reverse the value of test, changing it from true to false, or from false to true.

2.5.5 Conditional Operator

Any good programming language has some nifty little features that aren’t really necessary but that let you feel cool when you use them. Java has the conditional operator. It’s a ternary operator—that is, it has three operands—and it comes in two pieces, ? and :, that have to be used together. It takes the form

h*boolean-expression* i ? h*expression1* i : h*expression2* i

The computer tests the value of hboolean-expressioni. If the value is true, it evaluates hexpression1i; otherwise, it evaluates hexpression2i. For example:

next = (N % 2 == 0) ? (N/2) : (3\*N+1);

will assign the value N/2 to next if N is even (that is, if N % 2 == 0 is true), and it will assign the value (3\*N+1) to next if N is odd. (The parentheses in this example are not required, but they do make the expression easier to read.)

2.5.6 Assignment Operators and Type Conversion

You are already familiar with the assignment statement, which uses the symbol “=” to assign the value of an expression to a variable. In fact, = is really an operator in the sense that an assignment can itself be used as an expression or as part of a more complex expression. The value of an assignment such as A=B is the same as the value that is assigned to A. So, if you want to assign the value of B to A and test at the same time whether that value is zero, you could say:

52 *CHAPTER 2. NAMES AND THINGS*

if ( (A=B) == 0 )...

Usually, I would say, don’t do things like that!

In general, the type of the expression on the right-hand side of an assignment statement must be the same as the type of the variable on the left-hand side. However, in some cases, the computer will automatically convert the value computed by the expression to match the type of the variable. Consider the list of numeric types: byte, short, int, long, float, double. A value of a type that occurs earlier in this list can be converted automatically to a value that occurs later. For example:

int A;

double X;

short B;

A = 17;

X = A; // OK; A is converted to a double

B = A; // illegal; no automatic conversion

// from int to short

The idea is that conversion should only be done automatically when it can be done without changing the semantics of the value. Any int can be converted to a double with the same numeric value. However, there are int values that lie outside the legal range of shorts. There is simply no way to represent the int 100000 as a short, for example, since the largest value of type short is 32767.

In some cases, you might want to force a conversion that wouldn’t be done automatically. For this, you can use what is called a type cast. A type cast is indicated by putting a type name, in parentheses, in front of the value you want to convert. For example,

int A;

short B;

A = 17;

B = (short)A; // OK; A is explicitly type cast

// to a value of type short

You can do type casts from any numeric type to any other numeric type. However, you should note that you might change the numeric value of a number by type-casting it. For example, (short)100000 is -31072. (The -31072 is obtained by taking the 4-byte int 100000 and throwing away two of those bytes to obtain a short—you’ve lost the real information that was in those two bytes.)

When you type-cast a real number to an integer, the fractional part is discarded. For example, (int)7.9453 is 7. As another example of type casts, consider the problem of get ting a random integer between 1 and 6. The function Math.random() gives a real number between 0.0 and 0.9999. . . , and so 6\*Math.random() is between 0.0 and 5.999. . . . The type cast operator, (int), can be used to convert this to an integer: (int)(6\*Math.random()). Thus, (int)(6\*Math.random()) is one of the integers 0, 1, 2, 3, 4, and 5. To get a number between 1 and 6, we can add 1: “(int)(6\*Math.random()) + 1”. (The parentheses around 6\*Math.random() are necessary because of precedence rules; without the parentheses, the type cast operator would apply only to the 6.)

The type char is almost an integer type. You can assign char values to int variables, and you can assign numerical constants in the range 0 to 65535 to char variables. You can also use explicit type-casts between char and the numeric types. For example, (char)97 is ’a’, (int)’+’ is 43, and (char)(’A’ + 2) is ’C’.

∗ ∗ ∗

*2.5. DETAILS OF EXPRESSIONS* 53

Type conversion between String and other types cannot be done with type-casts. One way to convert a value of any type into a string is to concatenate it with an empty string. For example, "" + 42 is the string "42". But a better way is to use the function String.valueOf(x), a static member function in the String class. String.valueOf(x) returns the value of x, converted into a string. For example, String.valueOf(42) is the string "42", and if ch is a char variable, then String.valueOf(ch) is a string of length one containing the single character that is the value of ch.

It is also possible to convert certain strings into values of other types. For example, the string "10" should be convertible into the int value 10, and the string "17.42e-2" into the double value 0.1742. In Java, these conversions are handled by built-in functions.

The standard class Integer contains a static member function for converting from String to int. In particular, if str is any expression of type String, then Integer.parseInt(str) is a function call that attempts to convert the value of str into a value of type int. For example, the value of Integer.parseInt("10") is the int value 10. If the parameter to Integer.parseInt does not represent a legal int value, then an error occurs.

Similarly, the standard class Double includes a function Double.parseDouble. If str is a String, then the function call Double.parseDouble(str) tries to convert str into a value of type double. An error occurs if str does not represent a legal double value. ∗ ∗ ∗

Getting back to assignment statements, Java has several variations on the assignment operator, which exist to save typing. For example, “A += B” is defined to be the same as “A = A + B”. Every operator in Java that applies to two operands, except for the relational operators, gives rise to a similar assignment operator. For example:

x -= y; // same as: x = x - y;

x \*= y; // same as: x = x \* y;

x /= y; // same as: x = x / y;

x %= y; // same as: x = x % y;

q &&= p; // same as: q = q && p; (for booleans q and p)

The combined assignment operator += even works with strings. Recall that when the + operator is used with a string as one of the operands, it represents concatenation. Since str += x is equivalent to str = str + x, when += is used with a string on the left-hand side, it appends the value on the right-hand side onto the string. For example, if str has the value “tire”, then the statement str += ’d’; changes the value of str to “tired”.

2.5.7 Precedence Rules

If you use several operators in one expression, and if you don’t use parentheses to explicitly indicate the order of evaluation, then you have to worry about the precedence rules that deter mine the order of evaluation. (Advice: don’t confuse yourself or the reader of your program; use parentheses liberally.)

Here is a listing of the operators discussed in this section, listed in order from highest precedence (evaluated first) to lowest precedence (evaluated last):

Unary operators: ++, --, !, unary -, unary +, type-cast Multiplication and division: \*, /, %

Addition and subtraction: +, -

Relational operators: <, >, <=, >=

Equality and inequality: ==, !=

54 *CHAPTER 2. NAMES AND THINGS*

Boolean and: &&

Boolean or: ||

Conditional operator: ?:

Assignment operators: =, +=, -=, \*=, /=, %=

Operators on the same line have the same precedence. When operators of the same precedence are strung together in the absence of parentheses, unary operators and assignment operators are evaluated right-to-left, while the remaining operators are evaluated left-to-right. For example, A\*B/C means (A\*B)/C, while A=B=C means A=(B=C). (Can you see how the expression A=B=C might be useful, given that the value of B=C as an expression is the same as the value that is assigned to B?)

2.6 Programming Environments

Although the Java language is highly standardized, the procedures for creating, compil ing, and editing Java programs vary widely from one programming environment to another. There are two basic approaches: a command line environment, where the user types com mands and the computer responds, and an integrated development environment (IDE), where the user uses the keyboard and mouse to interact with a graphical user interface. While there is just one common command line environment for Java programming, there are several common IDEs, including Eclipse, NetBeans, and BlueJ. I cannot give complete or definitive information on Java programming environments in this section, but I will try to give enough information to let you compile and run the examples from this textbook. (Readers are strongly encouraged to read, compile, and run the examples. Source code can be downloaded from the book’s web page, http://math.hws.edu/javanotes.)

One thing to keep in mind is that you do not have to pay any money to do Java programming (aside from buying a computer, of course). Everything that you need can be downloaded for free on the Internet.

2.6.1 Java Development Kit

The basic development system for Java programming is usually referred to as the JDK (Java Development Kit). It is a part of Java SE, the Java “Standard Edition” (as opposed to Java EE for servers or Java ME for mobile devices). Note that Java SE comes in two versions, a Development Kit version (the JDK) and a Runtime Environment version (the JRE). The Runtime can be used to run Java programs, but it does not allow you to compile your own Java programs. The Development Kit includes the Runtime but also lets you compile programs. You need a JDK for use with this textbook.

Java was developed by Sun Microsystems, Inc., which is now a part of the Oracle corporation. Oracle makes the JDK for Windows, Mac OS, and Linux available for free download at its Java Web site. Many Windows computers come with a Java Runtime already installed, but you might need to install the JDK. Some versions of Linux come with the JDK either installed by default or on the installation media. Mac OS does not currently come with Java pre-installed. If you need to download and install the JDK, be sure to get the JDK for Java 7, Java 8, or later. As of summer, 2014, it can be downloaded from

http://www.oracle.com/technetwork/java/javase/downloads/index.html

*2.6. PROGRAMMING ENVIRONMENTS* 55

If a JDK is properly installed on your computer, you can use the command line environment to compile and run Java programs. An IDE will also require a JDK, but it might be included with the IDE download.

2.6.2 Command Line Environment

Many modern computer users find the command line environment to be pretty alien and unin tuitive. It is certainly very different from the graphical user interfaces that most people are used to. However, it takes only a little practice to learn the basics of the command line environment and to become productive using it.

To use a command line programming environment, you will have to open a window where you can type in commands. In Windows, you can open such a command window by running the program named cmd. (In Windows 7, click “Start / Program Files / Accessories / Command Prompt.” In Windows 8, press the Windows and X keys together to bring up the “Power User Menu,” and select “Command Prompt.”) In Mac OS, you want to run the Terminal program, which can be found in the Utilities folder inside the Applications folder. In Linux, there are several possibilities, including an old program called xterm; try looking for “Terminal” in your applications menu.

No matter what type of computer you are using, when you open a command window, it will display a prompt of some sort. Type in a command at the prompt and press return. The computer will carry out the command, displaying any output in the command window, and will then redisplay the prompt so that you can type another command. One of the central concepts in the command line environment is the current directory which contains files that can be used by the commands that you type. (The words “directory” and “folder” mean the same thing.) Often, the name of the current directory is part of the command prompt. You can get a list of the files in the current directory by typing in the command dir (on Windows) or ls (on Linux and Mac OS). When the window first opens, the current directory is your home directory, where all your files are stored. You can change the current directory using the cd command with the name of the directory that you want to use. For example, to change into your Desktop directory, type in the command cd Desktop and press return.

You should create a directory (that is, a folder) to hold your Java work. For example, create a directory named javawork in your home directory. You can do this using your computer’s GUI; another way to do it is to open a command window, cd to the directory that you want to contain the new dirctory, and enter the command mkdir javawork. When you want to work on programming, open a command window and use the cd command to change into your work directory. Of course, you can have more than one working directory for your Java work; you can organize your files any way you like.

∗ ∗ ∗

The most basic commands for using Java on the command line are javac and java; javac is used to compile Java source code, and java is used to run Java stand-alone applications. If a JDK is correctly installed on your computer, it should recognize these commands when you type them in on the command line. Try typing the commands java -version and javac -version which should tell you which version of Java is installed. If you get a message such as “Command not found,” then Java is not correctly installed. If the “java” command works, but “javac” does not, it means that a Java Runtime is installed rather than a Development Kit. (On Windows, after installing the JDK, you need to modify the Windows PATH environment variable to make this work. See the JDK installation instructions on Oracle’s download site for information about

56 *CHAPTER 2. NAMES AND THINGS*

how to do this.)

To test the javac command, place a copy of TextIO.java into your working directory. (If you downloaded the Web site of this book, you can find it in the directory named source; you can use your computer’s GUI to copy-and-paste this file into your working directory. Alternatively, you can navigate to TextIO.java on the book’s Web site and use the “Save As” command in your Web browser to save a copy of the file into your working directory.) Type the command:

javac TextIO.java

This will compile TextIO.java and will create a bytecode file named TextIO.class in the same directory. Note that if the command succeeds, you will not get any response from the computer; it will just redisplay the command prompt to tell you it’s ready for another command.

To test the java command, copy a sample program such as Interest2.java from this book’s source directory into your working directory, or download it from the web site. First, compile the program with the command

javac Interest2.java

Remember that for this to succeed, TextIO must already be in the same directory. Then you can execute the program using the command

java Interest2

Be careful to use just the name of the program, Interest2, with the java command, not the name of the Java source code file or the name of the compiled class file. When you give this command, the program will run. You will be asked to enter some information, and you will respond by typing your answers into the command window, pressing return at the end of the line. When the program ends, you will see the command prompt, and you can enter another command. (Note that “java TextIO” would not make sense, since TextIO does not have a main() routine, and so it doesn’t make sense to try to execute it as a program.)

You can follow a similar procedure to run all of the examples in this book. Some examples require additional classes, such as TextIO, in addition to the main program. Remember to place any required classes in the same folder as the program that uses them. (You can use either the .java or the .class files for the required classes.)

∗ ∗ ∗

To create your own programs, you will need a text editor. A text editor is a computer program that allows you to create and save documents that contain plain text. It is important that the documents be saved as plain text, that is without any special encoding or formatting information. Word processor documents are not appropriate, unless you can get your word processor to save as plain text. A good text editor can make programming a lot more pleasant. Linux comes with several text editors. On Windows, you can use notepad in a pinch, but you will probably want something better. For Mac OS, you might download the free TextWrangler application. One possibility that will work on any platform is to use jedit, a good programmer’s text editor that is itself written in Java and that can be downloaded for free from www.jedit.org.

To work on your programs, you can open a command line window and cd into the working directory where you will store your source code files. Start up your text editor program, such as by double-clicking its icon or selecting it from a Start menu. Type your code into the editor window, or open an existing source code file that you want to modify. Save the file into your working directory. Remember that the name of a Java source code file must end in “.java”, and the rest of the file name must match the name of the class that is defined in the file. Once the file is saved in your working directory, go to the command window and use the javac command

*2.6. PROGRAMMING ENVIRONMENTS* 57

to compile it, as discussed above. If there are syntax errors in the code, they will be listed in the command window. Each error message contains the line number in the file where the computer found the error. Go back to the editor and try to fix one or more errors, save your changes, and then try the javac command again. (It’s usually a good idea to just work on the first few errors; sometimes fixing those will make other errors go away.) Remember that when the javac command finally succeeds, you will get no message at all. Then you can use the java command to run your program, as described above. Once you’ve compiled the program, you can run it as many times as you like without recompiling it.

That’s really all there is to it: Keep both editor and command-line window open. Edit, save, and compile until you have eliminated all the syntax errors. (Always remember to save the file before compiling it—the compiler only sees the saved file, not the version in the editor window.) When you run the program, you might find that it has semantic errors that cause it to run incorrectly. In that case, you have to go back to the edit/save/compile loop to try to find and fix the problem.

2.6.3 Eclipse

In an Integrated Development Environment, everything you need to create, compile, and run programs is integrated into a single package, with a graphical user interface that will be familiar to most computer users. There are a number of different IDEs for Java program development, ranging from fairly simple wrappers around the JDK to highly complex applications with a multitude of features. For a beginning programmer, there is a danger in using an IDE, since the difficulty of learning to use the IDE, on top of the difficulty of learning to program, can be overwhelming. However, for my own programming, I generally use the Eclipse IDE, and I introduce my students to it after they have had some experience with the command line. I will discuss Eclipse in some detail and two other IDEs, NetBeans and BlueJ, in much less detail. All of these IDEs have features that are very useful even for a beginning programmer, although a beginner will want to ignore many of their advanced features.

You can download an Eclipse IDE from eclipse.org. It is a free program. Eclipse is itself written in Java. It requires a Java Runtime Environment, but not necessarily a JDK, since it includes its own compiler. You should make sure that the JRE or JDK, Version 7 or higher is installed on your computer, as described above, before you install Eclipse. There are several versions of the Eclipse IDE; you can use the “Eclipse IDE for Java Developers.”

The first time you start Eclipse, you will be asked to specify a workspace, which is the directory where all your work will be stored. You can accept the default name, or provide one of your own. When startup is complete, the Eclipse window will be filled by a large “Welcome” screen that includes links to extensive documentation and tutorials. You can close this screen, by clicking the “X” next to the word “Welcome”; you can get back to it later by choosing “Welcome” from the “Help” menu.

The Eclipse GUI consists of one large window that is divided into several sections. Each section contains one or more views. For example, a view can be a text editor, it can be a place where a program can do I/O, or it can contain a list of all your projects. If there are several views in one section of the window, then there will be tabs at the top of the section to select the view that is displayed in that section. Each view displays a different type of information. The whole set of views is called a perspective. Eclipse uses different perspectives, that is different sets of views of different types of information, for different tasks. For compiling and running programs, the only perspective that you will need is the “Java Perspective,” which is the default. As you become more experienced, you might want to the use the “Debug Perspective,” which

58 *CHAPTER 2. NAMES AND THINGS*

has features designed to help you find semantic errors in programs.

The Java Perspective includes a large area in the center of the window that contains text editor views. This is where you will create and edit your programs. To the left of this is the Package Explorer view, which will contain a list of your Java projects and source code files. To the right are some other views that I don’t find very useful, and I suggest that you close them by clicking the small “X” next to the name of each one. Several other views that will be useful appear in a section of the window below the editing area. If you accidently close one of the important views, such as the Package Explorer, you can get it back by selecting it from the “Show View” submenu of the “Window” menu. You can also reset the whole window to its default contents by selecting “Reset Perspective” from the “Window” menu.

∗ ∗ ∗

To do any work in Eclipse, you need a project. To start a Java project, go to the “New” submenu in the “File” menu, and select the “Java Project” command. In the window that pops up, it is only necessary to fill in a “Project Name” for the project and click the “Finish” button. The project name can be anything you like. The project should appear in the “Package Explorer” view. Click on the small triangle or plus sign next to the project name to see the contents of the project. Assuming that you use the default settings, there should be a directory named “src,” which is where your Java source code files will go. It also contains the “JRE System Library”; this is the collection of standard built-in classes that come with Java.

To run the TextIO based examples from this textbook, you must add the source code file TextIO.java to your project. If you have downloaded the Web site of this book, you can find a copy of TextIO.java in the source directory. Alternatively, you can navigate to the file on-line and use the “Save As” command of your Web browser to save a copy of the file onto your computer. The easiest way to get TextIO into your project is to locate the source code file on your computer and drag the file icon onto the project name in the Eclipse window. If that doesn’t work, you can try using copy-and-paste: Right-click the file icon (or control-click on Mac OS), select “Copy” from the pop-up menu, right-click the project’s src folder in the Eclipse window, and select “Paste”. (Be sure to paste it into the src folder, not into the project itself; files outside the source folder are not treated as Java source code files.) Another option is to add the file directly to the src folder inside your workspace directory. However, Eclipse will not automatically recognize a file added in this way; to make Eclipse find the file, right-click the project name in the Eclipse window and select “Refresh” from the pop-up menu. In any case, TextIO should appear under “src” in your project, inside a package named “default package”. Once a file is in this list, you can open it by double-clicking it; it will appear in the editing area of the Eclipse window.

To run any of the Java programs from this textbook, copy the source code file into your Eclipse Java project in the same way that you copied TextIO.java. To run the program, right click in the editor window, or on the file name in the Package Explorer view (or control click in Mac OS). In the menu that pops up, go to the “Run As” submenu, and select “Java Application”. The program will be executed. If the program writes to standard output, the output will appear in the “Console” view, in the area of the Eclipse window below the editing area. If the program uses TextIO for input, you will have to type the required input into the “Console” view—click the “Console” view before you start typing, so that the characters that you type will be sent to the correct part of the window. (For an easier way to run a program, find and click the small “Run” button in Eclipse’s tool bar.) Note that when you run a program in Eclipse, it is compiled automatically. There is no separate compilation step.

You can have more than one program in the same Eclipse project, or you can create addi-

*2.6. PROGRAMMING ENVIRONMENTS* 59

tional projects to organize your work better. Remember to place a copy of TextIO.java in any project that requires it.

∗ ∗ ∗

To create a new Java program in Eclipse, you must create a new Java class. To do that, right-click the Java project name in the “Project Explorer” view. Go to the “New” submenu of the popup menu, and select “Class”. (Alternatively, there is a small icon in the toolbar at the top of the Eclipse window that you can click to create a new Java class.) In the window that opens, type in the name of the class that you want to create. The class name must be a legal Java identifier. Note that you want the name of the class, not the name of the source code file, so don’t add “.java” at the end of the name. Examples in this book use the “default package,” so you will also want to erase the contents of the box labeled “Package.” (See the last section of this section for more information about packages.) Finally, click the “Finish” button to create the class. The class should appear inside the “src” folder, in the “default package,” and it should automatically open in the editing area so that you can start typing in your program.

Eclipse has several features that aid you as you type your code. It will underline any syntax error with a jagged red line, and in some cases will place an error marker in the left border of the edit window. If you hover the mouse cursor over the error marker or over the error itself, a description of the error will appear. Note that you do not have to get rid of every error immediately as you type; some errors will go away as you type in more of the program. If an error marker displays a small “light bulb,” Eclipse is offering to try to fix the error for you. Click the light bulb—or simply hover your mouse over the actual error—to get a list of possible fixes, then double click the fix that you want to apply. For example, if you use an undeclared variable in your program, Eclipse will offer to declare it for you. You can actually use this error-correcting feature to get Eclipse to write certain types of code for you! Unfortunately, you’ll find that you won’t understand a lot of the proposed fixes until you learn more about the Java language, and it is not a good idea to apply a fix that you don’t understand—often that will just make things worse in the end.

Eclipse will also look for spelling errors in comments and will underline them with jagged red lines. Hover your mouse over the error to get a list of possible correct spellings. Another essential Eclipse feature is content assist. Content assist can be invoked by typing Control-Space. It will offer possible completions of whatever you are typing at the moment. For example, if you type part of an identifier and hit Control-Space, you will get a list of identifiers that start with the characters that you have typed; use the up and down arrow keys to select one of the items in the list, and press Return or Enter. (You can also click an item with the mouse to select it, or hit Escape to dismiss the list.) If there is only one possible completion when you hit Control-Space, it will be inserted automatically. By default, Content Assist will also pop up automatically, after a short delay, when you type a period or certain other characters. For example, if you type “TextIO.” and pause for just a fraction of a second, you will get a list of all the subroutines in the TextIO class. Personally, I find this auto-activation annoying. You can disable it in the Eclipse Preferences. (Look under Java / Editor / Content Assist, and turn off the “Enable auto activation” option.) You can still call up Code Assist manually with Control-Space.

Once you have an error-free program, you can run it as described above. If you find a problem when you run it, it’s very easy to go back to the editor, make changes, and run it again. Note that using Eclipse, there is no explicit “compile” command. The source code files in your project are automatically compiled, and are re-compiled whenever you modify them.

60 *CHAPTER 2. NAMES AND THINGS*

2.6.4 NetBeans

Another IDE for professional programming is NetBeans. It can be downloaded from net beans.org. Alternatively, a bundle containing both NetBeans and the JDK is available on Oracle’s Java download page.

Using NetBeans is very similar to using Eclipse. Even the layout of its window is very similar to the Eclipse window. Create a project in NetBeans with the “New Project” command. You will have to select the type of project in a pop-up window. You want to create a “Java Application.” The project creation dialog will have a suggested name for the project, which you will want to change. It also has an option to create a main class for the project, which is selected by default. If you use that option, you should change the class name. For use with this book, the name should not be in a “package”; that is, it should not include a period.

A project will have a “Source Folder” where the source code files for the project are stored. You can drag TextIO.java and other files onto that folder, or you can copy-and-paste them from the file system. For running a file, you can right-click the file and select “Run File” from the pop-up menu. There is also a “Run” button in the NetBeans toolbar. There is no explicit compilation step. Input and ouput are done in an area below the edit window, just as in Eclipse.

When you are editing a file, NetBeans will mark errors as you type. (Remember, again, that many errors will go away on their own as you continue to type.) If NetBeans displays an error marker with a light bulb in the left-hand margin of the editor, you have to click the light bulb to get a list of possible automatic fixes for the error. NetBeans also has a “Code Completion” feature that is similar to Content Assist in Eclipse. Just press Control-Space as you are typing to get a list of possible completions.

2.6.5 BlueJ

Finally, I will mention BlueJ, an IDE that is designed specifically for people who are learning to program. It is much less complex than Eclipse or NetBeans, but it does have some features that make it useful for education. BlueJ can be downloaded from bluej.org.

In BlueJ, you can begin a project with the “New Project” command in the “Project” menu. A BlueJ project is simply a folder. When you create a project, you will have to select a folder name that does not already exist. The folder will be created and a window will be opened to show the contents of the folder. Files are shown as icons in the BlueJ window. You can drag .java files from the file system onto that window to add files to the project; they will be copied into the project folder as well as shown in the window. You can also copy files directly into the project folder, but BlueJ won’t see them until the next time you open the project. For example, you can do this with TextIO.java and the sample programs from this book. When you restart BlueJ, it should show the last project you were working on, but you can open any project with a command from the “Project” menu.

There is a button in the project window for creating a new class. An icon for the class is added to the window, and a .java source code file is created in the project folder. The file is not automatically opened for editing. To edit a file, double-click its icon in the project window. An editor will be opened in a separate window. (A newly created class will contain some default code that you probably don’t want; you can erase it and add a main() routine instead.) The BlueJ editor does not show errors as you type. Errors will be reported when you compile the program. Also, it does not offer automatic fixes for errors. It has a less capable version of Eclipse’s Content Assist, which seems only to work for getting a list of available subroutines in a class or object; call up the list by hitting Control-Space after typing the period following the

*2.6. PROGRAMMING ENVIRONMENTS* 61

name of a class or object.

An editor window contains a button for compiling the program in the window. There is also a compile button in the project window, which compiles all the classes in the project. To run a program, it must already be compiled. Right-click the icon of a compiled program. In the menu that pops up, you will see “void main(String[] args)”. Select that option from the menu to run the program. Just click “OK” in the dialog box that pops up. A separate window will open for input/output.

One of the neatest features of BlueJ is that you can actually use it to run any subroutine, not just main. If a class contains other subroutines, you will see them in the list that you get by right-clicking its icon. A pop-up dialog allows you to enter any parameters required by the routine, and if the routine is a function, you will get another dialog box after the routine has been executed to tell you its return value. This allows easy testing of individual subroutines. Furthermore, you can also use BlueJ to create new objects from a class. An icon for the object will be added at the bottom of the project window, and you can right-click that icon to get a list of subroutines in the object. This will, of course, not be useful to you until we get to object-oriented programming in Chapter 5.

2.6.6 The Problem of Packages

Every class in Java is contained in something called a package. Classes that are not explicitly put into a package are in the “default” package. Almost all the examples in this textbook are in the default package, and I will not even discuss packages in any depth until Section 4.5. However, some IDEs force you to pay attention to packages.

In fact, the use of the default package is discouraged, according to official Java style guide lines. Nevertheless, I have chosen to use it, since it seems easier for beginning programmers to avoid the whole issue of packages, at least at first. If Eclipse or NetBeans tries to put a class into a package, you can delete the package name from the class-creation dialog to get it to use the default package instead. But if you do create a class in a package, the source code starts with a line that specifies which package the class is in. For example, if the class is in a package named test.pkg, then the first line of the source code will be

package test.pkg;

In an IDE, this will not cause any problem unless the program you are writing depends on TextIO. A class that is in a non-default package cannot use a class from the default package. To make TextIO available to such a class, you can move TextIO to a named, non-default package. This means that the source code file TextIO.java has to be modified to specify the package: A package statement like the one shown above must be added to the very beginning of the file, with the appropriate package name. (The IDE might do this for you, if you drag TextIO.java from the default package into a non-default package.) If you add TextIO to the same package as the class that uses it, then TextIO will be automatically available to that class. If TextIO is in a different named package, you have to add an “import” statement to the other class to make TextIO available to it. For example, if TextIO is in the package textio, add the statement

import textio.TextIO;

to the top of the other source code file, just after its own package declaration. By the way, if you use packages in a command-line environment, other complications arise. For example, if a class is in a package named test.pkg, then the source code file must be in a subdirectory named “pkg” inside a directory named “test” that is in turn inside your main Java

62 *CHAPTER 2. NAMES AND THINGS*

working directory. Nevertheless, when you compile or execute the program, you should be in the main directory, not in a subdirectory. When you compile the source code file, you have to include the name of the directory in the command: Use “javac test/pkg/ClassName.java” on Linux or Mac OS, or “javac test\pkg\ClassName.java” on Windows. The command for executing the program is then “java test.pkg.ClassName”, with a period separating the package name from the class name. However, you will not need to worry about any of that when working with almost all of the examples in this book.

Exercises 63 Exercises for Chapter 2

1. Write a program that will print your initials to standard output in letters that are nine lines tall. Each big letter should be made up of a bunch of \*’s. For example, if your initials were “DJE”, then the output would look something like:

\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*

\*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\*\*\*\*\*\*\*

\*\* \*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\* \*\*

\*\* \*\* \*\* \*\* \*\*

\*\*\*\*\* \*\*\*\* \*\*\*\*\*\*\*\*\*\*

2. Write a program that simulates rolling a pair of dice. You can simulate rolling one die by choosing one of the integers 1, 2, 3, 4, 5, or 6 at random. The number you pick represents the number on the die after it is rolled. As pointed out in Section 2.5, the expression

(int)(Math.random()\*6) + 1

does the computation to select a random integer between 1 and 6. You can assign this value to a variable to represent one of the dice that are being rolled. Do this twice and add the results together to get the total roll. Your program should report the number showing on each die as well as the total roll. For example:

The first die comes up 3

The second die comes up 5

Your total roll is 8

3. Write a program that asks the user’s name, and then greets the user by name. Before outputting the user’s name, convert it to upper case letters. For example, if the user’s name is Fred, then the program should respond “Hello, FRED, nice to meet you!”.

4. Write a program that helps the user count his change. The program should ask how many quarters the user has, then how many dimes, then how many nickels, then how many pennies. Then the program should tell the user how much money he has, expressed in dollars.

5. If you have N eggs, then you have N/12 dozen eggs, with N%12 eggs left over. (This is essentially the definition of the / and % operators for integers.) Write a program that asks the user how many eggs she has and then tells the user how many dozen eggs she has and how many extra eggs are left over.

A gross of eggs is equal to 144 eggs. Extend your program so that it will tell the user how many gross, how many dozen, and how many left over eggs she has. For example, if the user says that she has 1342 eggs, then your program would respond with

Your number of eggs is 9 gross, 3 dozen, and 10

64 *CHAPTER 2. NAMES AND THINGS*

since 1342 is equal to 9\*144 + 3\*12 + 10.

6. This exercise asks you to write a program that tests some of the built-in subroutines for working with Strings. The program should ask the user to enter their first name and their last name, separated by a space. Read the user’s response using TextIO.getln(). Break the input string up into two strings, one containing the first name and one containing the last name. You can do that by using the indexOf() subroutine to find the position of the space, and then using substring() to extract each of the two names. Also output the number of characters in each name, and output the user’s initials. (The initials are the first letter of the first name together with the first letter of the last name.) A sample run of the program should look something like this:

Please enter your first name and last name, separated by a space. ? *Mary Smith*

Your first name is Mary, which has 4 characters

Your last name is Smith, which has 5 characters

Your initials are MS

7. Suppose that a file named “testdata.txt” contains the following information: The first line of the file is the name of a student. Each of the next three lines contains an integer. The integers are the student’s scores on three exams. Write a program that will read the information in the file and display (on standard output) a message that contains the name of the student and the student’s average grade on the three exams. The average is obtained by adding up the individual exam grades and then dividing by the number of exams.